A Software System Generator

For Data Structures

by

Martin J. Sirkin

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

Doctor of Philosophy

University of Washington

1994

Approved by _____
(Chairperson of Supervisory Committee)

_____

_____

_____

_____

Program Authorized
to Offer Degree _____

Date _____

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature _____

Date _____

University of Washington

Abstract

A Software System Generator For Data Structures

by Martin J. Sirkin

Chairperson of the Supervisory Committee: Professor David Notkin
Department of Computer Science and Engineering

Although data structures are a fundamental part of most applications, using and writing data structures is time-consuming, difficult, and error-prone. Programmers often select inappropriate data structures for their applications because they do not know which data structure to use, they do not know how to implement a particular data structure, or they do not have an existing implementation of the data structure to use.

This dissertation describes a model and a technology for overcoming these problems. Our approach is based on *non-traditional parameterized types* (NPTs). NPTs are an extension to *traditional parameterized types* (TPTs), which are already familiar to most programmers. Our NPTs are based on the GenVoca domain modeling concepts, vertical parameterization, a consistent high-level interface, and a transformational compiler.

Our research has led to the construction of a software system generator for data structures called Predator. Predator is able to transform data structure declarations and data structure-independent functions into efficient code. Predator also allows programmers to adjust a data structure's implementation by simply changing its declaration and recompiling.

This dissertation discusses our model (and how it differs from standard models), our Predator compiler, and the results of our validation efforts.

Table of Contents

List of Figures

List of Tables

Acknowledgments

So many to thank, and such a small space...

First, and most importantly, I want to thank my wonderful wife, Jeanne. Without her, I never would have completed this dissertation. Not only did she provide love and encouragement, she's given me three of the best reasons a man has ever had for going home each night. She's also a great technical editor. Thanks, love.

And I want to thank Dirk, Julian, and Rachel. You have made my life so complete and my days so very, very long. But I wouldn't trade it for anything in the world. Taking care of you three is more fun than anything I've ever done before.

I also want to thank Don Batory. His guidance throughout this research, his insights into computer science and software engineering, his prodding, his patience, and (most importantly) his friendship have all been invaluable. I will always appreciate what you've done for me.

I want to also thank several people at UW. I want to thank all of the members of my committee, particularly David Notkin, for their careful reading of my dissertation. Their insightful suggestions have improved the dissertation greatly. I also want to thank Frankye Jones, the only other Susan Butcher fan I know. She has put up with altogether too many registration snafus with my name on them. I really appreciate all that she's done.

Within the Predator group at UT, I thank all those students who shared their ideas and their time with me. Special thanks to Vivek Singhal and Jeff Thomas for innumerable and invaluable research discussions. I just wish that you two didn't like Mexican food so much.

Finally, I want to thank my family, particularly my mother and father, for their support and encouragement over all these many years. I learned to love learning from my parents and brothers, and I can think of no greater gift I could have received.

Dedication

I dedicate this dissertation to my father, Julius H. Sirkin. Dad, I just wish you could have been here to see this work completed.

Chapter 1

Introduction

1.1 The burden of data structure programming

Designing, writing, and debugging programs is a time-intensive task. Of the different aspects of writing programs of moderate to large complexity, implementing data structures often consumes a disproportionately large portion of a programmer's time. Given the pervasiveness of data structures in software, it is not surprising that there have been a number of attempts to automate the generation of data structure code. Further, since this task is fairly general, many of the proposed solutions for automation have come from different research communities, such as reuse, software components, object orientation, transformation systems, and persistent object bases [Kru92, Par83, Lam91]. While many of the proposed solutions have merit, most have concentrated on one area of software engineering, thereby not including useful features from others.

Eliminating the drudgery of programming data structures is clearly an important, but still unsolved, problem. Our thesis is that a practical solution rests on a software component technology that integrates concepts of parameterized types, standardized interfaces, domain modelling, and compiler optimizations [Bat92a, Sir93, Pri91]. Integration of these concepts is both technically challenging and a fertile area of research. More importantly, it leads to a technology for assembling complex data structures from pre-written and standardized components.

The key to a successful component technology for data structures, and reusable software components in general, is choosing an appropriate set of abstractions. If the abstractions are not fundamental, the likelihood of producing reusable components is reduced. There are several basic technical reasons why we believe that other well-trodden approaches to constructing software components - in particular, data structures - will ultimately not succeed. In this

thesis, we explain our views on the limitations of traditional approaches, as well as our proposed solutions to overcome their limitations.

A common problem is that programmers often hard-code their choice of data structures into the design of their programs. This makes it difficult to later modify the data structure and its code. (Modifications are often needed to add features and enhance performance). This is a problem of maintenance and evolution [Hor84]. Some projects [Sno89] have features which specifically aid program maintenance and evolution, but which also make other phases of programming more difficult. Our goal is to provide a standardized, high-level interface for our components to increase reuse and allows for simpler system evolution. In addition, we envision a mechanism to extend component interfaces to simplify program maintenance. This approach is orthogonal to that taken by others [Sno89, Gar92], and allows us to add features, such as invariants and assertions, without affecting our overall model.

A further goal of our research is to be able to capture certain programming features which are not normally considered to be data structures (such as concurrency, inheritance, client/server storage, before and after actions, statistics gathering) in our model. A major portion of our effort has been to show how these non-data structure concepts are implementable, as simple components within our libraries of plug-compatible components.

A primary result of our research is a prototype system which allows programmers to generate code for complex data structures more easily and faster than previously possible. The prototype is called Predator, which is an acronym for *PREcompiler for DAta sTRuctures*. We believe that Predator is a first step towards a powerful and practical tool for writing data structures that will offer gains in software productivity and increased code efficiency.

1.2 A programming example

This section briefly describes a task which is data structure intensive. The basic structures and interconnections of the system will be described, as well as some of the basic operations to be performed. The reason for this is to demonstrate that even elementary tasks give rise to a large degree of data structure complexity. This explosion of complexity is a prime motivation for our approach of automatic code generation.

Consider a simulation of a supermarket (for exposition purposes, we will discuss only a portion of the effort needed for a full simulation). Suppose a supermarket is composed of the

following entities (elements): items, checkout lines, cash registers, customers, and employees. The following figure depicts some attributes each of the elements might possess:

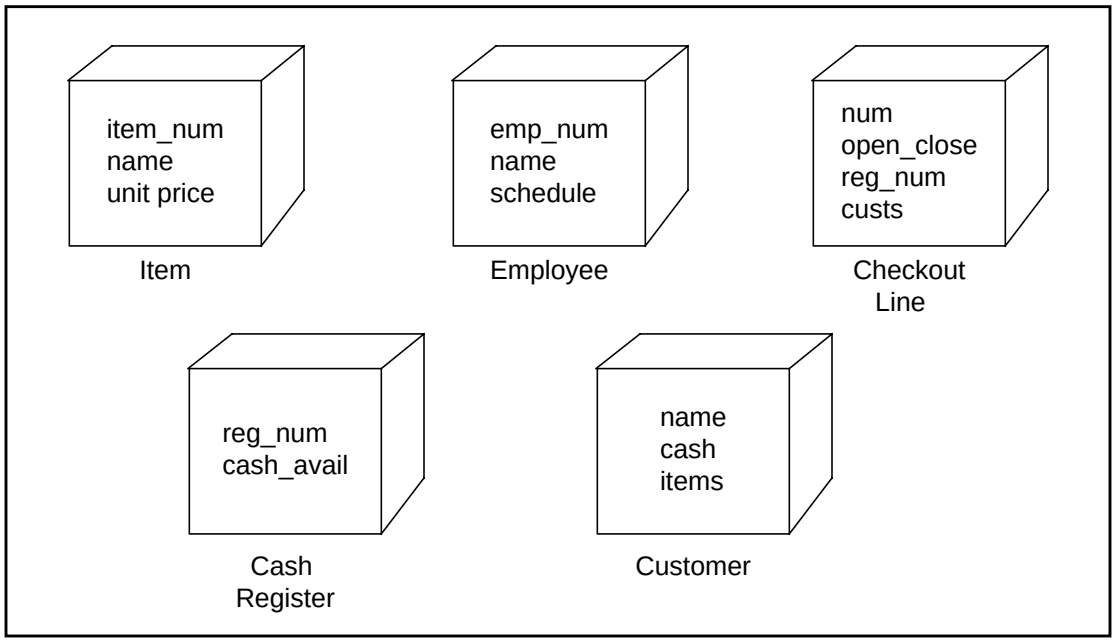

Figure 1.1: Supermarket elements

Now, consider the checkout area of the supermarket. A checkout area consists of a set of checkout lines, perhaps modeled as an array. Each of those lines must contain a set of customers, waiting to check out. A common implementation is to use a queue of customers. Customers check out one at a time, and are enqueued at the back of the line. Each customer has a basket of items. An item list might be stored as a simple linked list. Finally, employees must work the cash register at the head of each open line, and employees can only work the hours they are scheduled to work.

While this seems straightforward, a closer investigation reveals complicating considerations:

1. Customers may wish to exit from the middle of the queue, or add items to their basket while in the middle of the queue, both of which are difficult to express with standard queue semantics.

2. Join relationships must be maintained between cash registers and the employees currently manning them.

3. While most of the data can be stored in transient memory, certain data (such as items stocked at the store) must be stored persistently.

4. As our simulation evolves, we may wish to improve our employee scheduling algorithm, requiring a new, more efficient, data structure to be used for schedules. We would like to make this change with as little impact as possible (or none) to the existing code utilizing employee schedules.

Obviously, we could make this simulation far more complex. However, even this simple example demonstrates the complexity explosion of data structure programming. Managing this complexity is a significant undertaking.

1.3 Characteristics of an acceptable solution

A major focus point of this research has been to determine what properties are required for a tool to aid programmers in writing efficient data structures. We believe that such a tool should:

1. Allow programmers to code data structures faster and more easily than by hand,

2. Allow programmers to utilize efficient data structures, even if they do not fully understand how those data structures work,

3. Enable programmers to debug their programs without having to see or interact with the code generated by the system (i.e. debug at the algorithmic, not data structure, level),

4. Allow programmers to change the underlying data structure implementation in a program without having to modify the application code,

5. Generate code that is at least as efficient (in execution time) as that which could be generated by hand by the “average” programmer (i.e. within 10% of highly-tuned and hand-optimized code).

6. Aid the programmer in the evolution process by enabling the addition of new features and implementations without having to rewrite existing applications.

1.4 Potential solutions

As mentioned earlier, there have been many attempts to create tools which provide the functionality described in the previous section. In general, these attempts can be grouped into three categories: component libraries, object-oriented libraries, and transformation systems. Most of the solutions that we review straddle these rough categorizations; they are hybrids, as is our proposed system. A more detailed description of several of the more significant efforts can be found in Chapter 6. Each of the categories described below share one concept: the *software component* (or module, layer). The component is a fundamental unit which encapsulates a behavior, and which presents an external interface. Details such as implementation, extensibility and scalability may differ by category, but they all share components as their underlying basis.

1.4.1 Software component libraries

Component libraries are perhaps the most common solution to data structure programming. Component libraries can be found almost everywhere today, from stand-alone libraries [Boo87, Boo90, Gor90] to libraries that are included with compilers [Bor92] and operating systems [Lea88]. In a component library, many different data structures are represented, each by its own component in the library. These components are atomic and are treated as indivisible units of computation. Each component presents its interface to the outside world. Most often that interface is unique for each component, but this is not a requirement. Component libraries may or may not be object-oriented. Object-oriented component libraries tend to be structured as a forest of related trees of components, where each tree is structured by inheritance. Also, programmers who use object-oriented component libraries can easily create new components via inheritance.

An important feature of component libraries is that the code which implements the components is static and fixed. That is, the code for each component is written by the library author and is provided to the user. While the code may be heavily optimized, components cannot be modified to suit the context of the application without having to understand and modify the code of the original component.

Components are often parameterized to allow them to represent a family of related components. Most component libraries with which we are familiar provide some level of parameterization, but usually only for such things as bounds, ranges, and *generic instantiations*.

Generic instantiations allow one, for example, to parameterize a linked-list component to create a linked-list of customers, or a linked-list of bank accounts.

While components cannot be broken apart, they can often be composed. The composition facility in most libraries is fairly rigid, which limits the number and types of compositions allowed. Illustrative examples of this will be presented in Section 3.3.

1.4.2 Transformation systems

Another (broad) category of systems is so-called *transformation* systems. Transformation systems are characterized by the transformations they perform on components to create new components which are tailored to the data structure or environment desired. Examples of data structure transformation systems include [Sno89, Coh89, Coh93, Nov83, Nov92].

The chief feature which distinguishes transformation systems from library approaches is their ability to dynamically alter their components as needed. This flexibility allows them, in principle, to be more task-specific, and hence more efficient.

Transformation systems do not retain the notion of *atomic* (or indivisible) components. A component may be modified during transformation, based on meaning-preserving transformation rules. Transformation systems do not imply any particular compositional technique, nor do they restrain (as do OO approaches) the interfaces of the differing components.

1.5 An optimizing compiler approach

While many of the approaches described above are indeed useful, it is clear to us that no one approach is inherently better than the others; each has strengths and weaknesses. In fact, if one looks at commercially available data structure programming aids [Boo87, Lea88, Boo90, Lam91], no one approach seems to dominate over any other.

The subject of software architectures is gaining importance [Gar93]. Software architectures deal with the scalability issues of programming-in-the-large. GenVoca is but one approach to create scalable domain-specific software architectures [Hei90, Hei91, Cam92, Wei90]. It is a model of hierarchical software construction that has been used in software generators for a variety of rather disparate domains (databases [Bat88, Bat90], network communication [Hut91, Oma90], and avionics [Cog93]). Its key feature is that it exploits the maturity of domains by standardizing their fundamental abstractions, and then standardiz-

ing the way in which software systems of the domain are built from these abstractions. We will discuss GenVoca at greater length in Section 3.1 [Bat92b].

The relevance of GenVoca became evident when we realized that data structures are hierarchical; they can be understood as compositions of plug-compatible layers. Although GenVoca offered a basis for understanding how data structure code might be generated, it was our experience that the GenVoca concepts were insufficient, by themselves, to lead to a data structure programming environment. To achieve the goals we stated in Section 1.3, we feel that it is necessary to augment concepts from other disciplines. While those concepts will be fully explored later in this dissertation, a few will be mentioned here:

1. We have included a consistent, high level interface often found in persistent object bases [Lam91] and databases [Acm91, Bat88, Bat90, Dat83].

2. Our system relies heavily on compiler and transformation technology. Our components are always created dynamically to suit the particular data structures and operations needed.

3. Utilizing compiler technology also allows us to extensively explore the area of domain-specific optimizations. We feel that this is crucial for generating high-performance components.

4. We have extended the notion of component parameterization from classical work [Gog86] to allow components to be parameterized by fields, as well as the more standard constants, functions, and data types.

5. We have defined a series of *specializations* that allow us to customize components with new functions, and to add new functions without affecting existing components or application code. In addition, our specialization model provides a recursive macro processor, which allows components to augment their “standardized” interfaces.

The name “GenVoca” was given to the model that unified the designs of two software system generators: Genesis, a generator of database systems, and Avoca, a generator of network protocols. Just as our project developed from Genesis, another language/compiler-based project [Abb92] was spawned from Avoca. It is worth noting that this follow-on project to Avoca added a set of features to GenVoca that is almost identical to ours. We believe that this is a confirmation of our findings.

1.6 Goal

We have motivated the need for automatic generation of data structure code, and mentioned several approaches that have already been explored. We believe that the design and construction of an efficient, easy-to-use tool in this area is an important area of research. Such a tool, if designed and implemented, could be of tremendous use in both academia and industry.

Our research goals are (1) to identify the concepts that are needed for a scalable data structure generator, (2) to construct a compiler that validates these concepts, and (3) to conduct a series of experiments to demonstrate that our compiler satisfies the characteristics of an acceptable solution (discussed earlier in Section 1.3).

The next section describes the contents of this thesis in greater detail.

1.7 Overview

Chapter 2 introduces the concepts of traditional parameterized types (TPTs). The chapter describes TPTs, explains how classical solutions attempt to solve the data structure programming problem with TPTs, and details some of the inherent limitations of the TPT model as an acceptable solution.

In Chapter 3, we describe a variation of TPTs, which we call NPTs (non-traditional parameterized types). We begin by describing the GenVoca domain modelling concepts, on which NPTs are based. The individual features of NPTs are then described. Chapter 3 also defines how we express the data structure domain in terms of realms, includes a recapitulation of the identified limitations of TPTs, and shows how NPTs overcome these limitations.

We describe our prototype in chapter 4. Sections are included on terminology, a description of **DS** and **LINK** realms, the realm language syntax, the actual compiler and its environment, and examples of use. This chapter also includes many of the design decisions made during the implementation, and suggestions for future further implementations of NPTs.

As mentioned before, validation of the implementation was a crucial part of this effort. In chapter 5 we describe the methodology for validating our work, present a series of experiments that were conducted on the prototype, and report the results of the validation effort.

We compare and contrast our work to related work in chapter 6. Since our work is in a field which is defined by the intersections of several other disciplines, we also distinguish where we believe the boundaries of that intersection lie.

Chapter 7 presents an evaluation of this work. Sections are included discussing design decisions made, as well as implementation notes and concerns.

In chapter 8 we present a summary of our research and results. In addition, we offer our thoughts on future work in the area of data structure precompilers.

Chapter 2

Traditional parameterized types

2.1 Description

A *traditional parameterized type* (TPT) is a conventional expression of a generic, abstract data type. It is abstract in that the data type externalizes the interface, while hiding implementation details. A TPT is a data type that has been parametrically generalized to compactly represent a family of related types. A generic stack, for example, could be instantiated to form a stack of integers or a stack of strings. Generics in Ada [Ghe87, Coh90] and templates in C++ [Str91] are well-known implementations of TPTs.

In this section, we examine the current uses and assumptions of TPTs. We then expose fundamental limitations in the design of TPTs, current implementations of TPTs, and in the concept of TPTs itself.

Goguen [Gog86] has proposed a model of parameterized programming in which two distinctly different forms of parameterization were identified: horizontal and vertical. Horizontal parameterization is equivalent to TPTs: the *interface* of the component is parameterized by other data types (as well as by constants). Vertical parameterization corresponds to the notion of *layering*. That is, the *internal implementation* of the data type is parameterized by other data types. We find that while TPTs are rich in horizontal parameterization, they are lacking in vertical. We will describe our augmentation of TPTs with vertical parameterization in Chapter 3. The rest of this chapter will describe horizontal parameterization and composition with TPTs.

Below, we present the two common methods of TPT composition: instantiation of a base type and instantiation of another generic data type.

1. The most common use of generics is in the instantiation of the base type. In the example introduced above, a generic stack might be instantiated with integers at one time and with strings at another, thus creating two new types. Each of these types, in turn, can be instantiated into real data structures. Figure 2.1 shows a generic stack, which is instantiated into a stack of integers and a stack of strings. It further shows instances of each of these new types. Note that this horizontal parameterization does not affect the internal implementation of the stack, only the externalized interface.

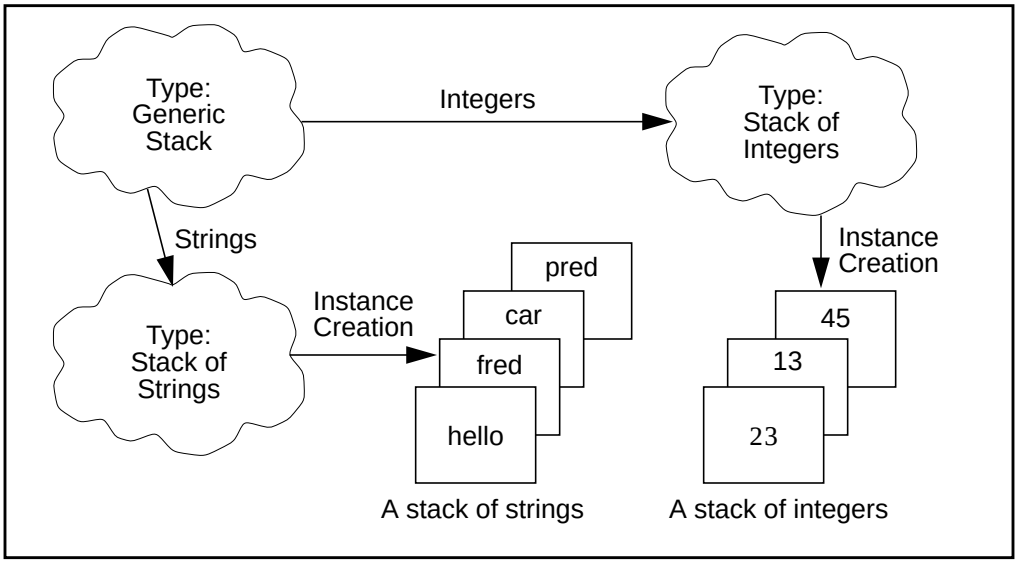

Figure 2.1: A stack of strings and a stack of integers instantiated from a generic stack

2. Generics can also be instantiated with other generics as their base type. An example of this type of instantiation is the composition of a list generic with a binary tree generic. This yields a list of trees in which each node of the list is a pointer to the root of a binary tree. That is, the base element type for each node of the list is the header structure of a binary tree (i.e. pointer to the root node). Figure 2.2 shows such a data structure.

   Note that the order of composition is significant. The composition of a binary tree of lists is completely different than that shown in Figure 2.2.

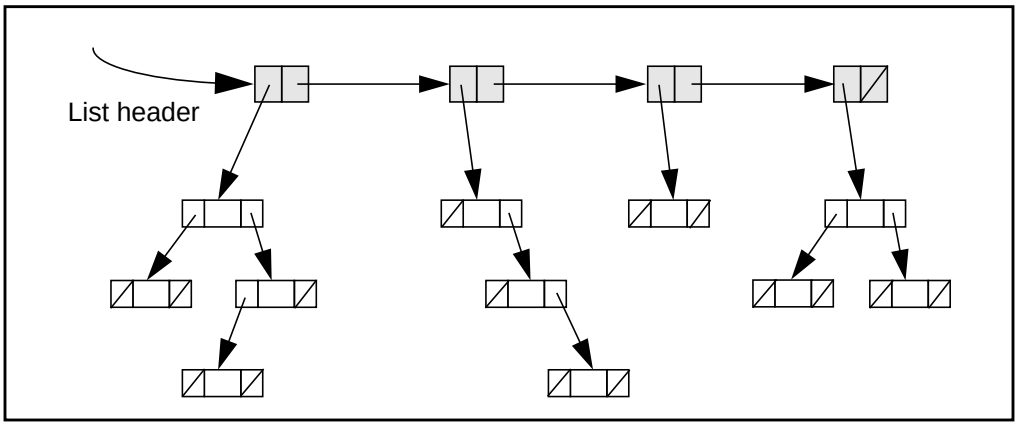

Figure 2.2: A traditional parameterized type: a list of trees

We will show in Section 3.3 that there are different kinds of generic abstractions. We call the one presented above *traditional parameterized types* (TPTs), because the basic concepts are well understood and established, even if they are not used as often as they should be in practice.

2.2 Limitations of TPTs

We have identified three general areas in which TPTs are deficient:

- Conceptual deficiencies (specialization, complex compositions, type transformations).
- Design deficiencies (ad-hoc interfaces, evolution, scalability).
- Implementation deficiencies (code efficiency)

Each of these limitations are elaborated in the following sections.

2.2.1 Difficulty of specialization

A TPT offers operations that its author believes are adequate for a wide variety of applications. However, in the context of a specific application, it is often the case that additional operations, unforeseen by the TPT author, are needed [Gar92, Sul92]. As the following examples illustrate, the penalty for non-extensible TPT interfaces can be severe inefficiencies or abandoning the use of the TPT altogether.

Consider a queue with the interface: *enqueue, dequeue, is_empty, is_full*. Suppose that it is necessary to delete items located in the interior of the queue. Using only the previously mentioned operations, one must dequeue each item from the queue, check to see if it is the requested element, and enqueue it if it is not. Typically the end of the loop is determined by either knowing the size of the queue or by placing a special end-marker on the queue before iterating. Clearly this is both inefficient and awkward, as it requires a dequeue and enqueue for each of the **n** items on the stack.

The situation is worse for stacks, whose interface is: *push, pop, is_empty, is_full*. It can be easily seen that, unlike queues, a second data structure (stack) is required for the deletion. The top item of the stack is popped. It is then pushed onto the second stack. When the item to be deleted is popped it is discarded. Then, each item from the second stack is popped, and pushed back onto the original stack (to retain the original ordering). Deleting an item in the interior of a stack is not only inefficient in time, but also in space.

Although the preceding examples may seem a bit contrived, they are not. Recall the checkout lines at a supermarket described in the introduction. One could easily model this as an array of queues. Using only operations provided by the queue interface, common events cannot be easily programmed. Consider the following:

- A customer in line realizes that he needs another item and leaves the middle of the line to do more shopping.
- A “10 items or fewer” line opens up. People leave other lines to queue up.
- Someone in a line “holds the place” for another person, who arrives, and enters the middle of the line.
- Every third customer in a given line is given a new promotional item (i.e. candy bar) while in line. (This is an example of modifying an element interior to a data structure.)

Each example clearly shows the need to *specialize* TPT definitions. Specialization is the process of modifying a type to augment its interface so that it conforms to what is needed by an application programmer. There are three alternatives, each of which has inherent flaws.

1. The TPT author tries to envision all specializations in advance and provides an expanded interface to cover all cases. This is difficult, if not impossible. Even if it could be accomplished, the approach often backfires - complex interfaces can be intimidating; programmers may choose not to use this expanded TPT by opting for simplicity. Note

that the “philosophy” of TPT design is to provide a clean and generic abstraction. Complex interfaces contradict this premise.

2. The author provides the TPT source, so that programmers can create their own specializations. This essentially nullifies many of the productivity advantages of TPTs, as programmers must now understand someone else's code, as well as the TPT module, and must program and debug the extensions as needed. Inheritance can be used to help reduce the specialization work required, but experience has shown that even this approach requires considerable additional programming work for customized applications [Tan89].

3. Don't use TPTs. In production level code, many component libraries are shipped in object form only. It is up to the programmer to either live with the TPT interface provided, or to not use TPTs at all.

None of these alternatives is satisfactory. We feel that components provided to a programmer must be able to be specialized easily and without much knowledge of the inner workings of the TPTs.

2.2.2 Complex compositions

It is widely believed that TPTs are the appropriate abstraction for encapsulating primitive data structures [Boo87, McN86a, McN86b, Pal90]. More complex structures are assumed to be created through TPT compositions. This is the case for the data structure depicted in Figure 2.2. In general, however, TPTs are far from adequate.

Many data structures that are used in practice are unlikely to be available in TPT libraries. A simple example is a data structure that simultaneously links its elements onto a binary tree (to maintain one ordering of the elements) and onto a linked list (to maintain a second ordering.) Figure 2.3 illustrates this structure. Each node contains pointers for both a binary tree and a linked list. Note that the root of the tree need not be the same as the head of the list.

It is important to recognize that the structure in Figure 2.3 *cannot* be created by parametric instantiation of the tree and linked list TPT module. Neither a list of trees nor a tree of lists is the same as the structure depicted in Figure 2.3.

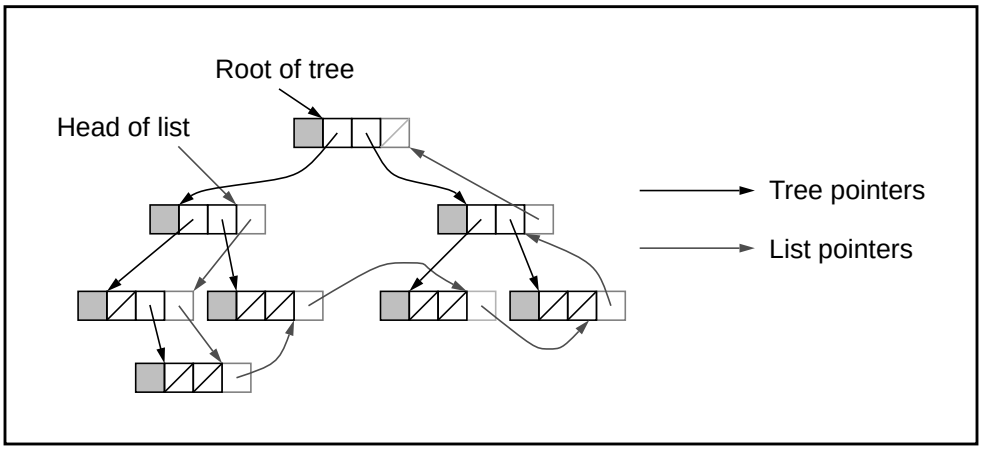

Figure 2.3: A binary tree/linked list data structure

There are TPT solutions for creating the structure shown above. For example, multiple inheritance can be used to capture the behavior of both the binary tree and the linked list. There are two potential problems with this approach. The first is a naming conflict problem[1]. The inherited characteristics may share names among data or functions[2]. The other problem is that, depending on the modules inherited, “glue code” may need to be written for the modules to inter-operate successfully. Glue code is programming added to integrate the code fragments from multiple components. An example of this might be a data structure with both a dynamic memory allocation scheme and an avail-list allocation scheme. The code inherited from these two modules are interrelated. That is, simply macro expanding the code from them (in arbitrary order) would not be sufficient. Languages such as SELF [Cha89] provide more sophisticated inheritance mechanisms, but still suffer from lack of knowledge about the domain (necessary to create merged, efficient components).

Mapping TPTs [Boo87] can be used to simulate (but not exactly match) the data structure shown in Figure 2.3, but “glue” code is still required, and the indirections from the mapping layer(s) incur additional (and possibly unacceptable) performance penalties.

Elements stored within a data structure seldom exist in isolation. Real systems often have elements in one structure *linked* to elements in other data structures (e.g. via pointers). Figure 2.4 shows a container of employees and a container of departments; each department

1. The name resolution problem is unresolved, with several possible solutions [Kai89].
2. This is particularly true in the case where the same component is inherited multiple times.

is connected to the employees in that department via a link. Consistency constraints, such as the existence (or nonexistence) of data items in other structures, are often coupled with such linkages.

In general, structures like Figure 2.3 and Figure 2.4 arise frequently in practice. Because TPTs offer little help in their construction, the TPT abstraction for encapsulating primitive data structures is not sufficient. Component composition is a basic operation of a data structure compiler. A fundamental goal of such a compiler is to automatically generate efficient "glue" code. Thus, we feel that TPTs without compilation and domain-specific knowledge optimizations are not sufficient to define complex data structures.

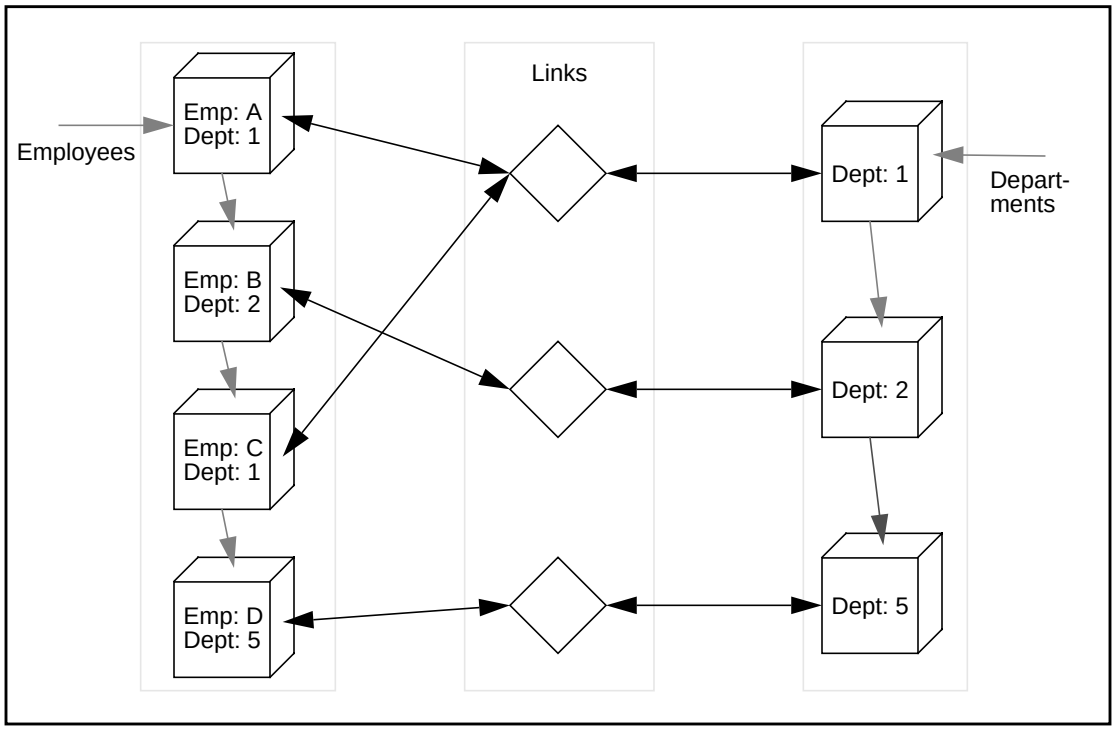

Figure 2.4: Two interlinked lists

2.2.3 Type transformations

Data structures can be modelled as mappings or type transformations; where an abstract type, devoid of any data structure implementation details, is mapped to a concrete type where the details of the transformation are now visible. For example, a linked list TPT adds the **`previous`** and **`next`** fields to each data element it stores, and introduces algorithms to maintain the consistency of the fields.

Inheritance is a simple, but common, method of realizing such transformations. That is, inheritance can add new fields and new operations to data types. However, there are many other type transformations that cannot be expressed in terms of new fields and/or operations. These include transformations where fields or operations are deleted, where the original data structure is partitioned into two or more sub-structures, and where the original fields are transformed into different types and/or sizes. Of these different type transformations, only the first two are native to TPT implementations. The others require additional capabilities.

Examples of algorithmic operations which require these more complicated type transformations include: segmentation, partitioning (fix sized), compression, and encryption (both fixed and variable sized). Because TPTs rely on mechanisms such as inheritance and mapping to transform types by field and operation addition, TPTs are unable to express certain type transformations as simple data structures, and thus cannot express important classes of data structures.

2.2.4 Ad hoc interfaces

Most TPTs have unique interfaces. This means that the TPT interface for lists is (typically) different from that of arrays, binary trees, etc. When an application program is written using a specific TPT, it is often difficult to change the underlying data structure (TPT) without triggering a substantial rewrite [Nov92]. While changing from a static array to a dynamic linked list may not be too difficult, going to a more complex structure entails considerable effort.

It is often the case that one does not understand all of the factors which affect the choice of data structures (TPTs) until well beyond the point where design changes can be easily made. If TPTs have unique interfaces, then it may be too expensive to retrofit a better suited data structure into existing code. Being able to change data structures “on the fly” (i.e. with minimal side effects) is highly desirable. The uniqueness of TPT interfaces actually inhibits experimentation with alternative implementations. It is often the case that a programmer’s first working solution to a problem is also his last.

2.2.5 Evolution and maintenance

TPTs by themselves do provide a small amount of support for program maintenance and system evolution [Gar92, Sul92]. Certainly, data hiding helps when maintaining a program, and having the ability to change the implementation of a module without having to rewrite the caller's program allows systems to evolve over time. However, TPTs, by themselves, do not provide further support for these important problems.

For example, declaring and maintaining invariants among data structures is useful when evolving a system. Programmers should have the ability to either declare that a relationship always be maintained by the TPT as an invariant, or (after manual insertions or deletions) to test by assertion a relationship with a TPT supplied interface. Although the latter functionality can be found in [Sno89] as part of the environment, the ability to declare and maintain invariants is generally absent from the TPT model.

2.2.6 Scalability

Current TPT libraries are populated with components that represent a unique combination of features that the TPT author believes are necessary. The Booch C++ Components [Boo90], for example, implements over 400 distinct data structures such as stacks, lists, hash tables, queues, and trees. The large number of components arises from feature combinatorics; components are differentiated according to their support for concurrency (e.g., sequential, guarded, concurrent, multiple), basic data structures (list, queues, stacks, etc.), space management (bounded vs. unbounded, managed vs. unmanaged), and features offered (iterator vs. noniterator, balking vs. nonbalking, etc.). Every legal combination of features yields a distinct data structure. Because there are many possible combinations, it is not surprising that this library is large.

Feature combinatorics are inherent to all libraries [Kru92]. Such libraries are inherently unscalable, simply because the number of features that programmers need is open-ended. Moreover, library components are typically written by hand, with occasional use of inheritance to minimize gross code replication. A methodology is required to allow the size of the component library to scale gracefully as new components are added[3]. The monolithic nature of

3. The problem of scalability is detailed further in [Bat93a].

TPT components makes this a difficult goal to attain.

2.2.7 Code efficiency

A standard method of implementing TPTs is to compile the code for each TPT component separately, with references and manipulations of generic objects performed via pointers [Ghe87]. This introduces an additional run-time overhead for resolving inter-TPT references. Another method of implementation is macro expansion, which is not always sufficient to provide efficient code [Nov92]. We feel that these composition mechanisms, by themselves, do not generate sufficiently efficient code.

In the domain of data structures, where performance is often critical, large run-time overheads or inefficient code are simply not acceptable.It is the nature of monolithic TPT components that they have no specialized knowledge either of other components, or how they will be composed with other components. This precludes them from being able to take advantage of *domain-specific knowledge*. Both partial evaluation and composition-based optimization (domain-specific) are mechanisms by which more efficient TPT implementations can be constructed.

Another problem with TPT implementations is that the TPT modules are written to be *general*. As we mentioned in Section 2.2.1, TPTs tend to implement every operation that the TPT author believes may be necessary. Often, a particular application needs only a fraction of the operations available, or only a portion of a particular operation. Since TPTs do not typically rely on compiler technology for dynamic instantiation, it is difficult, if not impossible to remove the undesired sections of the TPT from the main body of the component. This can result in an increase in program size, decrease in performance, or both.

2.3 Conclusions

The TPT (i.e. generic abstract data type) is an abstraction which is commonly believed to be an appropriate mechanism to solve a number of software engineering problems. While the concept of TPTs is useful, TPTs possess basic limitations which have not been fully appreciated by the software community. We feel that a more powerful mechanism is required. Our proposed solution is outlined in the next chapter.

Chapter 3

Non-traditional parameterized types

A *non-traditional parameterized type* (NPT) is a generalization of the TPT. It is based on the GenVoca model (Section 3.1), and Goguen's concept of vertical parameterization [Gog86].

Application writers use NPTs much as they use TPTs. Our NPTs provide a series of primitive functions (such as **`INSERT, DELETE, FOREACH...`**) which augment a high-level language (such as C). With NPTs, a programmer writes a program which consists of standard high-level code, as well as NPT function calls. NPTs also provide a mechanism for programmers to declare *type equations* (or annotations). Type equations specify the implementation of data structures used in the program. Finally, the programmer executes an NPT *precompiler*, which translates the NPT primitive function calls into efficient high-level code, based on the type equations. The resultant code is then compiled and linked with a standard language compiler and linker, producing an executable program.

In this chapter we introduce the GenVoca domain modelling approach, from which NPTs are derived. We define terminology necessary for NPTs, and discuss NPT concepts. We then explain how NPTs overcome the deficiencies of TPTs that we noted in Section 2.2.

The concepts, examples, and arguments in favor of NPTs presented in this chapter are programming-language independent. For clarity, however, we have chosen to code the examples in the C programming language. The reader should not, therefore, infer from this that NPTs are in any way dependent on the C programming language.

3.1 The GenVoca model

GenVoca is a domain-independent model for defining scalable families of hierarchical systems as compositions of reusable components. Other system generators [Hei90, Haa90] also possess organizations similar to GenVoca. As mentioned in Section 1.5, GenVoca is derived from two independently-conceived software system generators, Genesis [Bat88, Bat90] and Avoca [Hut91, Oma90]. The foundation for GenVoca has its roots in Parnas' families of systems [Par76], Habermann's FAMOS project [Hab76], and Goguen's model of parameterized programming [Gog86, Tra93]. GenVoca is characterized by realms (or libraries) of plug-compatible components, symmetric components, and type equations.

3.1.1 Components and realms

A hierarchical software system is defined by a series of progressively more abstract virtual machines. A *component* (or layer) is an implementation of a virtual machine. The set of components that implement the same virtual machine is called a *realm*.

The components of a realm can be enumerated. Consider realms **R** and **S**:

```
R = { a, b, c }

S = { d[x : R], e[x : R], f[x : R] }
```

The notation above signifies that realm **R** contains three members (**a**, **b**, and **c**). Each of these is a different implementation of the virtual machine for realm **R**. Realm **S** also contains three components (**d[]**, **e[]**, and **f[]**), which are distinct implementations of the virtual machine for realm **S**.

Components may be parameterized in terms of realms. For example, each component of realm **S** requires a single parameter of realm **R**. This means that each component of realm **S** exports the virtual machine interface of realm **S**, and imports the virtual machine interface of realm **R**.

Components can be understood as *transformations*. Component **d[]**, for example, can be seen as a transformation that maps objects and operations of virtual machine **S** to objects and operations of virtual machine **R**. The importance of this concept is that the transformation performed by **d[]** does *not depend on any specific implementation of* **R** (i.e. which component

of realm **R** is provided). In other words, component **d[]** encapsulates a complex mapping between the two realm interfaces.

3.1.2 Type equations

GenVoca systems are modeled as *type equations*. Consider the following two systems:

```
System_1 = d[b];
System_2 = f[a];
```

System_1 is a composition of **d[]** with **b**; **System_2** is a composition of **f[]** with **a**. Note that both systems are of type **S**. This means that both systems implement the same virtual machine and are interchangeable implementations of the interface of realm **S**. They are interchangeable in that both **System_1** and **System_2** provide the same interface, and usually perform the same work. Even in the (very small number of) cases where the two systems do not perform the same work, they still provide equivalently compileable programs. While this may seem to be a weak constraint, it turns out that defining systems in this manner is quite powerful and useful. In our experience, it is actually very difficult to construct two systems (of the same type) which do *not* perform the same work.

Realms and their components define a grammar whose sentences (i.e. composite compositions) are software systems. Adding a new component to a realm is akin to adding a new rule to the grammar; the family of systems is automatically enlarged. Because large families of systems can be built using relatively few components, the GenVoca model is a *scalable* method of software construction.

In principle, any component of realm **R** can be inserted into component **d[]** to instantiate it. However, there are always certain combinations of components that do not make sense, even though the type equation is well-formed. Examples of well-formed type equations that perform no useful work are provided later in this dissertation. A short example (from the data structure domain) would be a data structure such as:

```
delflag[malloc[]];
```

which is defined to do nothing more than *malloc* elements (which can be logically deleted) on the heap. In other words, elements are not interlinked in any way, and thus a **FOREACH** state-

ment would be unable to traverse through the elements in the data structure. Even so, the type equation would generate compileable code (even for the **FOREACH** statement).

In an automated system (such as our NPT compiler), additional domain-specific *design rules* can be used to preclude illegal or pointless combinations. In the example described above, it is somewhat trivial for an NPT compiler to note that there exists a conflict between the **FOREACH** operation and the type equation provided. The compiler can then return the appropriate warning or error to inform the programmer of the mismatch. Attribute grammars appear to be a unifying formalism that can be used to define realms, their components, and design rules [Bat92c].

3.1.3 Symmetry

Recursion, in the form of *symmetric* components, is fundamental to GenVoca. Symmetric components have the unusual property that they can be composed in arbitrary ways. A component is considered to be *symmetric* if it exports the same interface that it imports (i.e. a symmetric component of realm **T** has at least one parameter of type **T**). In the realm below, components **n[]** and **m[]** are symmetric whereas component **p** is not.

```
T = { n[x : T], m[x : T], p }
```

Because **n[]** and **m[]** are symmetric, compositions such as **n[m[p]]**, **m[n[p]]**, **n[n[p]]**, **m[m[p]]** are possible. Note that the latter two demonstrate that a symmetric component may be composed with itself. In general, the order in which components are composed is significant. It may affect the semantics, performance, and behavior of the resulting software system. Care must be taken that the system specified with a given type equation properly models the behavior for the desired software system.

3.2 Basic concepts

Before discussing the features of NPTs, we define some basic terms.

Elements are the basic building blocks from which data structures are built. NPT elements are nothing more than instances of structures (or records); they are the tuples stored in a data structure. For example, the customer elements shown in Figure 3.1 as individual figures might have the following definition:

```
struct customer
{
  char name[30];
  int age;
}
```

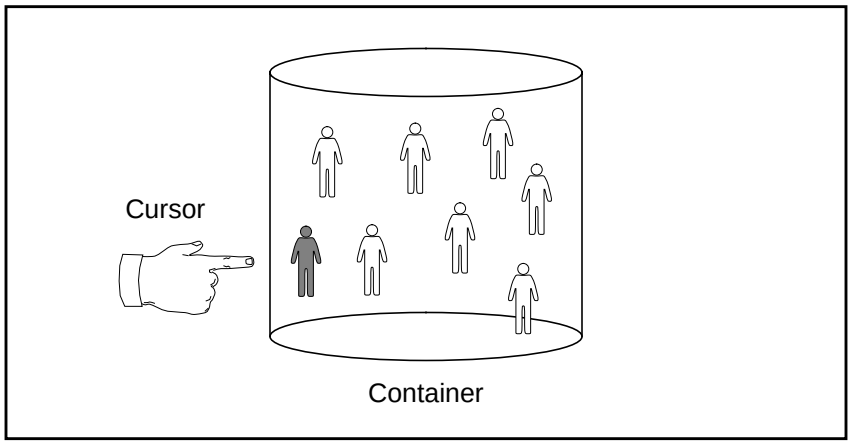

Figure 3.1: A container of customer elements

A container is a receptacle of elements of a particular base type. Typical data structures that programmers commonly use are nothing more than implementations of the container abstraction. The semantics of storage and retrieval (e.g. the container is an array of elements or a linked list of elements) will be discussed in a later section. The container depicted in Figure 3.1 holds eight customer elements.

Cursors are objects which allow programmers to access individual elements in a container. Cursors may be set to reference individual elements of a container, advanced to the next element in the container (or reversed to the previous element), or used to traverse each element in a container. The cursor shown in Figure 3.1 references the shaded customer element within the container. Cursors are completely *opaque*, in that the programmer has no idea of how a specific cursor is implemented.

Note that the only way in which elements can be retrieved, modified, or deleted from a container is through a cursor.

A *schema* is a description of a data structure (i.e. a container), including both the base element type of the data structure and implementation details. It is, in effect, a data type for a data structure. Each schema is tightly coupled to a single base element type. In addition, a schema defines both the implementation of the data structure (i.e. the layering), as well as the

implementation of the primitive functions that operate on the data structure. Schemas are declared as follows:

```
SCHEMA <schema_name> ON ELEMENT <base_element> = <implementation>;
```

where **<schema_name>** is the name of the schema definition, **<base_element>** is the name of a declared element (structure), and **<implementation>** is a valid type equation. Type equations are fundamental to our work, and were described in Section 3.1.1.

It is important to understand that schemas are not physical objects. Schemas are nothing more than templates that describe how an NPT data structure should be created. The actual data structure described in the schema does not exist until at least one container (of that schema type) is declared. NPTs do not restrict the number of container instantiations of a particular schema that a programmer may declare.

All of the entities listed above are declarative constructs. The following example declares a single schema which implements an array of customers, a container and a pointer to a container of that schema type, and a single cursor which traverses the declared container:

```
SCHEMA a_schema ON ELEMENT CUSTOMER = array[100];
a_schema a_cont, *ptr_cont;
CURSOR a_curs ON a_cont;
```

In the above example, we defined one schema type, **a_schema**. We then declared one container (data structure) of that schema type, **a_cont**, and a pointer variable which points to a container of that type, **ptr_cont**. Finally, we defined a cursor, **a_curs**, which is declared to range over elements in the **a_cont** container.

A *link* is a relational join which connects together elements from two containers. For example, a container of customers and a container of orders (of merchandise) could be linked together so that each customer is connected to all orders that he has placed. Links are specified via a relational *predicate*, which determines which pairs of cursors satisfy the link. Links are declared as follows:

```
LINK <link_name> ON <cardinality> <parent_container>
                 TO <cardinality> <child_container>
                 USING <implementation>
                 WHERE <link_predicate>;
```

where **cardinality** is either **ONE** or **MANY**, **<link_name>** is the name of the link, **<parent_container>** and **<child_container>** are declared containers, **<implementation>** is a valid **LINK** type equation, and **<link_predicate>** is a selection predicate, involving terms of the two containers, to qualify which elements in the two containers participate in the link.

A *composite cursor* is a construct used to traverse (one or more) links. Composite cursors provide the same function for links that cursors do for containers. A composite cursor is a set of *n* distinct container cursors which range over potentially distinct containers. Each container cursor used within a composite cursor is responsible for referencing an element in a specific container. Thus, a composite cursor's state can be represented by all of the container elements of the composite cursor. At any given moment, the container cursors of a composite cursor reference elements which satisfy the composite cursor's predicate. Composite cursors are defined in terms of a predicate, which is a conjunctive expression including expressions on cursors and link predicates. A composite cursor is declared as follows:

```
COMPCURS <cc_name> USING {<c1> <cont1>}+ WHERE <cc_pred>;
```

<cc_name> is the identifier of the composite cursor, and one cursor-container pair is declared for each container that participates in the composite cursor. The **<cc_pred>** is a selection predicate. The predicate may be composed of any combination of relational expressions on containers and links previously declared. For example, the composite cursor defined below will traverse pairs of employee and department elements, where the employee is the department's manager, and the employee is 64 years old:

```
LINK man_link ON ONE employee TO ONE dept
     WHERE employee.emp_id == dept.manager_id;

COMPCURS cc_man USING emp_curs employee dept_curs dept
         WHERE man_link && emp_curs.age == 64;
```

Composite cursors may be extended to range over as many different containers as necessary. Composite cursors may also be *self-referential*. Self-referential composite cursors declare at least two container cursors that range over the same container. For example, consider a container which stores employee records for a company. A self-referential composite cursor which returns all employees in the company, their department, and their manager, would be declared as follows:

```
LINK manager ON ONE employee TO ONE dept
     WHERE dept.manager == employee.emp_id;

LINK works_in ON ONE dept TO MANY employee
     WHERE dept.dept_num == employee.dept_num;

COMPCURS cc_manages USING e1 employee d1 dept e2 employee
     WHERE manager(e1, d1) && works_in(d1, e2);
```

To implement self-referential composite cursors, syntax must be added to each **<cc_pred>** clause to inform the compiler as to which container cursors are used for the clause. [1]

3.3 NPT features

In Chapter 2, we noted that TPTs implement *horizontal* parameterization. The GenVoca model, on the other hand, defines a set of layers that implement *vertical* parameterization. Our NPT model for data structures incorporates both horizontal and vertical parameterization. The following sections detail the specific augmentations we have made to the TPT model.

3.3.1 Realm partitioning

With our NPT model, we have chosen a broad decomposition of the data structure domain. All components which operate on elements within a single container are defined to belong to the *realm* of components called **DS** (for data structures). The **DS** realm includes most common data structures. Examples include arrays, lists, trees, and indexes. In addition, many other data transformations, which are not normally thought of as data structures, are representable as **DS** components. The complete list of **DS** components we have implemented can be found in Section 4.2.1.

```
DS = {array[], malloc[], persist[], delflag[x : DS], dlist[x : DS],
      bintree[x : DS], avail[x : DS]...}
```

Components within **DS** are defined to be either *terminals* or *non-terminals*. Terminal components are responsible for allocation and de-allocation of physical storage. Examples include array, malloc, and persist. All other components in **DS** are non-terminals. Non-terminal com-

1. Self-referential links are not supported in the current version of the compiler. However, Section 7.1 describes how to augment our system to support this feature.

ponents must be parameterized with at least one instance of another **DS** component (i.e, they are GenVoca symmetric components). For example, the component for unordered, doubly linked lists has the following definition:

```
dlist[x : DS] : DS
```

This component imports a virtual machine of type **DS,** and also exports the **DS** virtual machine. In other words, **dlist** is a simple non-terminal (symmetric) layer with no additional arguments.

In our model of data structures, we have defined one other realm called **LINK**. Members of **LINK** implement relational joins between two (and only two) containers. **LINK** components include join implementations such as nested loop and pointer-based links.

```
LINK = {linknest[x : LINK], linkpoint[x : LINK],
        linkterm[x : DS]...}
```

As with **DS**, the **LINK** realm implements both terminals and non-terminals. In the set above, **linknest** and **linkpoint** are symmetric components, whereas **linkterm** is not. The **LINK** realm is somewhat specialized, and not used as often as **DS**. For simplicity, in the sections that follow we will confine our general discussions to components within **DS**, even though the discussion applies to components of the **LINK** realm as well.

3.3.2 Consistent interface

One of the cornerstones of NPTs is that all components in a given realm share a *consistent interface*. Consistency is defined syntactically in that all components of a given realm implement exactly the same external interface (i.e. virtual machine). This consistency of interface is different than that of conventional data structure components. Different TPTs, for example, typically export different interfaces. With consistent interfaces, however, each member of realm **DS** exports exactly the same interface. Thus, the binary tree has the same external interface as the static array component, as does the timestamp component.

Consistent interfaces yield several benefits, chief among them being plug compatibility. Programmers may use any component (or valid composition of components) any place where a single **DS** component is called for. Perhaps the greatest advantage of consistent interfaces is that application programmers write to only one interface for all data structures. Thus, when new data structures are used by a programmer, no additional time is required to learn the

new data structure's interface. Debugging time is also greatly reduced. The result is a potentially large gain in programmer productivity.

Consistent interfaces also ease the burden of writing new components. Component writers for NPT components know, in advance, what functions they need to write. In addition, they can, in many cases, scavenge code from other components to help in writing new components.

The major problem brought on by consistent interfaces is that TPT-like functions are lost. For example, there is no inherent *push* or *pop* operators for a stack component. These component-specific functions are easily regained with NPTs with the use of a macro language which will be described in Section 4.1.3. Combining macros with a domain-specific optimizer (see Section 3.3.6) produce TPT-specific functions with competitive performance. Thus, application programmers attain the benefits of both plug compatibility and TPT-like behavior.

3.3.3 Vertical parameterization

NPTs utilize a compositional method different than that of TPTs. NPT data structure declaration is accomplished via type equations. While the syntax of type equations is not dissimilar from that of TPTs, the semantics are quite different (see Section 2.1). Each component defined in an NPT type equation adds (deletes, or modifies) fields to the base element structure, resulting in a new, more concrete element type. The new type is more concrete in that it contains more of the fields that implement the data structure in the type equation than did the original base element type. Resolving a type equation, therefore, is the process of applying successive transformations to the user-defined element type to create the element type used by the target data structure.

Thus, a long and complex type equation is responsible for generating one and only one data structure - not a composition as defined with TPTs. The following description of the transformations involved is somewhat more formal.

Let **P** be a program and **C** be a container that is referenced by **P**. We will write this as **P(C)**. **P** refers to **C** using the generic cursor operations outlined in Table 3.1. This means that **P** is data structure generic - i.e., it is not dependent on the data structure implementation of **C**.

Now suppose **C** is mapped, via a type equation, to a container **C′** which exposes all or part of the implementation of **C**. Concomitantly, **P** must be transformed into a program **P′**

that operates on **C′** and preserves the semantics of **P(C)**. Thus, plugging in a data structure as a (possibly partial) implementation of **C** transforms **P(C)** to **P′(C′)**.

It follows that a data structure layer (component) for containers is a pair of functions, where **C** is the domain of containers and **P** is the domain of programs (g_K: **C**$\rightarrow$**C**, g_π: **P**$\rightarrow$**P**). g_K is a *container mapping function* which transforms an abstract container **C** into a concrete (or less abstract) container **C′**. g_π is a *program mapping function* which deterministically transforms a program **P** into a corresponding program **P′**. The following are examples of container and programming mapping functions.

Consider a component for unordered lists ($LIST_K$: **C**$\rightarrow$**C**, $LIST_\pi$: **P**$\rightarrow$**P**). $LIST_K$ is a container mapping function. It links together all objects of the input container onto an unordered list. Figure 3.2a shows a container **C** with six customer objects. Figure 3.2b shows the resulting container $LIST_K$(**C**). This container has exactly the same objects as **C**, with the addition that each object has a **next** attribute that is used to reference the next object in the list. The container itself is also augmented with the attribute **head** to reference the head of the list.[2]

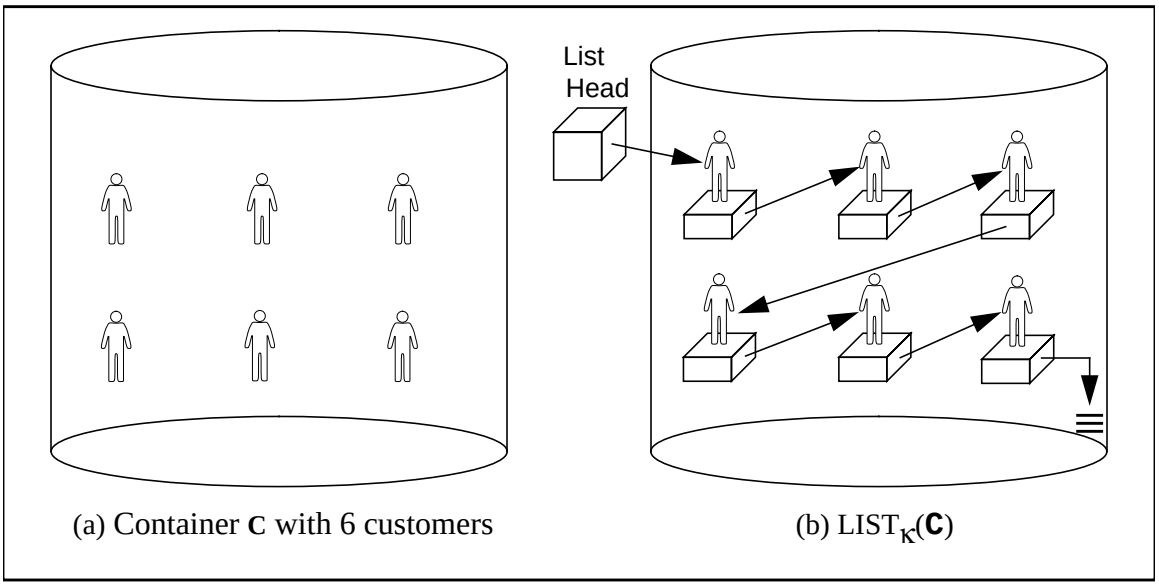

Figure 3.2: A Simple Container Transformation

2. Note that the order in which objects are linked onto containers reflects the order in which objects were inserted. This ordering is defined by the $LIST_\pi$ function. Thus, there is a unique container that results from a $LIST_K$ mapping.

$LIST_\pi$: **P**$\rightarrow$**P** is the corresponding program mapping function. $LIST_\pi$ replaces each operation on **C** with the corresponding code fragment that operates on $LIST_\kappa$(**C**). For example, an insertion into **C** is mapped to an insertion into $LIST_\kappa$(**C**) followed by a linking of the object onto a list. That is, the operation:

```
INSERT(C, obj, curs);
```

of program **P** is mapped to:

```
INSERT(K, obj, curs);
UPDATE(curs, Next, K.head);
K.head = curs.CURSOR_POS;
```

of program **P'** where κ is the container $LIST_\kappa$(**C**). Writing transformations for other cursor operations on **C** for LIST is straightforward.

Now consider the component for binary trees: ($BINTREE_\kappa$: **C,A**$\rightarrow$**C**, $BINTREE_\pi$: **P,A**$\rightarrow$**P**) where **A** is the domain of attributes for key fields. $BINTREE_\kappa$ is a container mapping function. Given a container and a key field, $BINTREE_\kappa$ produces a container where all objects of the input container are linked together onto a binary tree. Each object in **C** is transformed by the addition of two fields (`left` and `right`) that are needed to maintain binary tree linkages. The container itself is also augmented with the attribute `root` to reference the root of the binary tree. Figure 3.3 shows the mapping of **C** to $BINTREE_\kappa$(**C**, **A**) where **A** is an attribute of the objects in **C.**

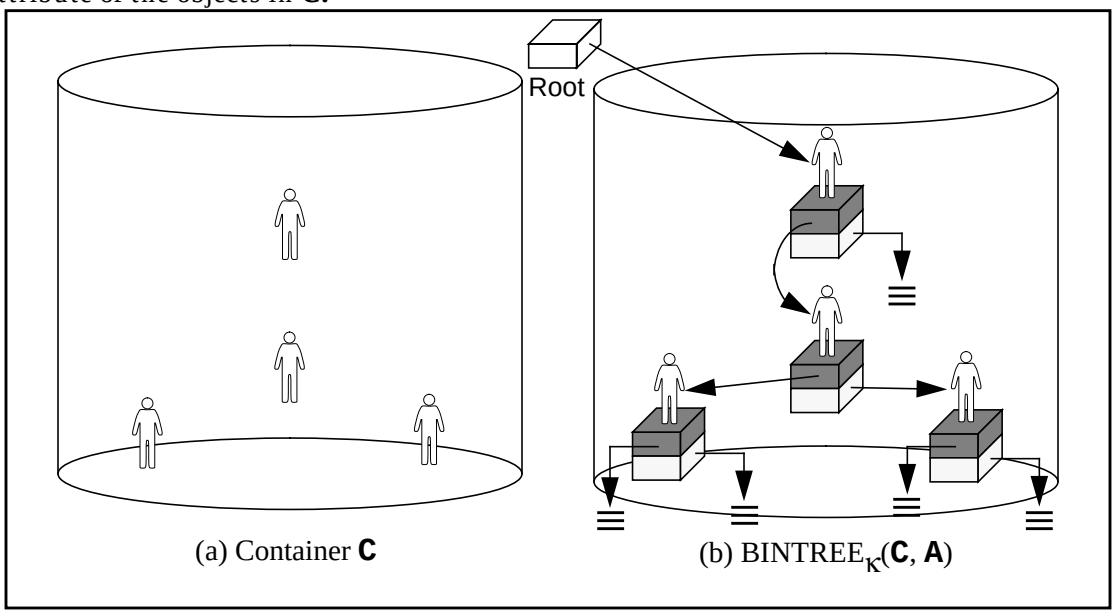

Figure 3.3: Binary Tree Transformation

$BINTREE_{\pi}$ is the corresponding program mapping function. It transforms operations on **C** to operations on $BINTREE_{K}$(**C**, **A**). As an example, inserting an object into **C** is mapped to an insertion of the object into $BINTREE_{K}$(**C**, **A**) followed by a linking of the object into the binary tree.

A key feature of this component abstraction is the symmetry of their mappings: containers are mapped to containers and programs are mapped to programs; the standard container interface remains invariant with respect to component transformations. This means that many different combinations and permutations of components are possible; each yields a different data structure and its support algorithms.

Recall that Figure 2.3 depicts an implementation of container **C** that is implemented by a composition of the binary tree and list components: $LIST_{K}(BINTREE_{K}(\mathbf{C}))$. First, the binary tree fields **`left`** and **`right`** are adorned onto each object of **C**, and then each adorned object is augmented with the list field **`next`**. The resulting program $LIST_{\pi}(BINTREE_{\pi}(\mathbf{P}))$ transforms an object insertion in **C** into an insertion into container $LIST_{K}(BINTREE_{K}(\mathbf{C}))$, a link of the object onto the list, and then a linkage of the object onto the binary tree.

NPTs, therefore, deal with program transformations. Programmers write data-structure independent algorithms, and specify container implementations as compositions of components. The transformation of programs and data types are done automatically through the use of in-line expansion and partial evaluation. More details on transformational processes can be found in [Par83, Gri90, Bal85, Coh93].

Containers are first-class objects. Thus, the more traditional forms of composition, such as that in Figure 2.2 can easily be supported. That is, a list of trees would be represented as follows: a container, which is a binary tree of elements would first be declared. Then a second container, which used the first container as its primitive element type would be declared. The declarations for this example would be:

```
SCHEMA bin_tree ON ELEMENT customer = bintree[malloc[]];
SCHEMA composition ON ELEMENT bin_tree = dlist[malloc[]];
```

The combination of both the (vertical parameterization) compositional semantics of NPTs and support for more traditional methods of compositions allows for a very large set of NPT data structures.

3.3.4 High-level interface

Because all the components in a given realm share the same interface, it is important that the interface be robust.

Our interface for **DS** largely reflects work performed in databases [Bat88, Bat90, Kor91] and persistent object bases [Lam91]. It is high-level, in that the interface provides little more than basic relational operations. The following table lists the primary operations of the **DS** realm interface:

Table 3.1: **DS** realm interface

Function Call	**Meaning**
INSERT(k, o, c [, h])	Insert object *o* into container *k*. Cursor *c* is an output parameter which is positioned on *o* in *k*. *h* is an optional hint about where to place object *o* (i.e., AT_END, AT_FRONT, AFTER or BEFORE (the position indicated by cursor *c* that has been positioned previously). If no hint is supplied, the layer semantics determine the positioning of the new object[a].
DELETE(c)	Delete the object referenced by cursor *c*.
UPDATE(c, a, v)	Assigns the value *v* to the attribute *a* of the object referenced by cursor *c*[b].
c.a = v	Same as UPDATE(c, a, v).
RESET(c, h)	Repositions cursor *c* either to the start of the container or to the end (based on the *h* argument).
ADVANCE(c)	Repositions cursor *c* on the next qualified object in *c*'s container. A status code is set in the cursor to GOOD if the advance succeeds, EOR otherwise.
REVERSE(c)	Repositions *c* to the previous qualified object in the container. The status code is set as in ADVANCE.
c++	Same as ADVANCE(c).
c--	Same as REVERSE(c).
LAST_CURS_OP(*c*)	Return the status of the last cursor operation on the cursor *c* (EOR or GOOD).
c.a	Return the value of attribute *a* of the object referenced by cursor *c*.
FOREACH(c) {code}	Execute the code fragment *code* for each object that can be referenced by cursor *c*. *c* is reset to the start of the container and is iterated through the container.
FIND(c, p)	Position cursor *c* to the next object that satisfies *c*'s predicate and the additional predicate *p*. The status code is set as in ADVANCE.

Table 3.1: **DS** realm interface

Function Call	**Meaning**
GETREC(c, o)	Retrieve the object referenced by cursor *c* and place it into the buffer specified by *o*.
c.CURSOR_POS	Return the location (untyped pointer) of the object referenced by cursor *c*.
c.CURSOR_POS = v	Position cursor *c* on the object with location *v*.
SWAP(c1, c2)	Swap the objects referenced by cursors *c1* and *c2*. Both cursors are referencing objects in the same container.

a. Note that **INSERT** will not override the semantics of a data structure. For example, if a programmer attempts to insert an element **AT_FRONT** of an ordered layer, the element will placed at the "correct" location, even if it is not the front. The compiler we have implemented will warn the programmer of this possible semantic violation at compile time.

b. Synthesized attributes (such as a next field for a linked list) which are added by the NPT compiler are not visible to the application programmer, and may not be altered with functions such as **UPDATE**.

All layers and composition of layers in **DS** implement the interface of Table 3.1. However, not all layers actually implement code for each of the functions of the interface. For example, the **size** layer is responsible for keeping track of the size of a given container (i.e. how many elements are stored in the container). The **advance** function is responsible for moving a cursor from one element in a container to the **next** element of the container. Since the **size** layer is only concerned with the size of the container, and not the interconnections among the elements, it is unable to assist in the **advance** operation. Thus, it generates no code for the **advance** function. In a similar vein, the **swap** operation generates no code for layers that are ordered. The actual mechanics of code generation will be discussed in Section 3.3.5.

The other realm we have defined is **LINK**. Links are responsible for connecting together two elements from one or more containers, based on a declared predicate. As with the **DS**

realm, all layers of the **LINK** realm share a common interface. That interface is described in the table below:

Table 3.2: **LINK** realm syntax

Function call	**Meaning**
FOREACH(*cc*) {*code*}	For each n-tuple of container cursors of the composite cursor *cc*, execute the code fragment specified by *code*.
RESET(*cc*)	Reset the composite cursor *cc* to the first n-tuple of container cursors for *cc*.
ADVANCE(*cc*)	Advance the composite cursor *cc* to the next n-tuple of container cursors for *cc*.
REVERSE(*cc*)	Move the composite cursor *cc* to the previous n-tuple of container cursors for *cc*.
LAST_CURS_OP(*cc*)	Return the status of the last cursor operation on the composite cursor *cc* (EOR or GOOD).

Containers and cursors are first-class objects. They may be stored as variables, passed as arguments, and de-referenced via pointers. This allows programmers to treat and manipulate data structures in much the same manner as any other variable in their program.

3.3.5 Compiler environment

We implemented our NPT model as a multi-pass compiler, with a scanner and parser, a symbol table, semantic tree, optimizer, and code generator. The compiler's main task involves reading and storing data structure descriptions, and transforming the primitive functions (such as insert) into valid source code that implements the function based on the type equation of the container.

In performing these transformations, the compiler performs three major functions: validation (type, name, and bounds checking), generation of "glue" code to allow for seamless composition of the different layers, and optimization to improve the performance of the generated NPT code. Optimization is important, and will be discussed in Section 3.3.6.

One of the main tasks of the NPT compiler is to verify and validate the programmer's use of data structures. Name and type verification is performed on all objects. The compiler also warns of data structures declared, but not used, as well as data structures referenced that

were not declared. Most importantly, the compiler notifies the programmer of operations performed on data structures where the operation is legal, but semantically invalid.

Simple components of the **DS** realm (such as array and list) can be composed via macro expansion. The code for each component is independent, and combining the two components requires no additional effort. More complex layers, however, present difficulties. Layers that modify the original element's structure often require the compiler to insert additional code which "glues" together the code from different layers. We have defined several NPT layers that either change the original element fields, remove them entirely, or move the fields into new and different data structures.

An example of this is the segment layer. The segment layer partitions the original container into two (or more) new containers. Thus, each element that is added to the (abstract) segmented container is broken up into two (or more) sub-elements. Whereas the programmer is dealing with the original element at the conceptual level, the compiler must now generate and integrate code for multiple data structures each time a primitive function is evaluated. For example, consider the following schema:

```
struct customer
{
  char f_name[20];
  char l_name[20];
  int age;
  char addr[50];
}

SCHEMA test_seg ON ELEMENT customer =
       segment[dlist[malloc[]],
               l_name,
               array[100],
               age,
               malloc[]];
```

This rather complex expression requires that each element of the declared data structure be broken up into three different sub-elements, each of which is stored in a separate data structures. So, for example, if we added the following customer:

```
{f_name = Jack, l_name = Smith, age = 44, addr = 55 Shady Lane}
```

to the segmented container, the first two fields would be added to the primary segment (container), the age field would be added to the secondary segment, and the address to a third. The

primary sub-element would then have pointers to the other sub-elements, and would be linked to other elements in the primary segment.

Figure 3.4 depicts such a segmented container, with two elements stored.

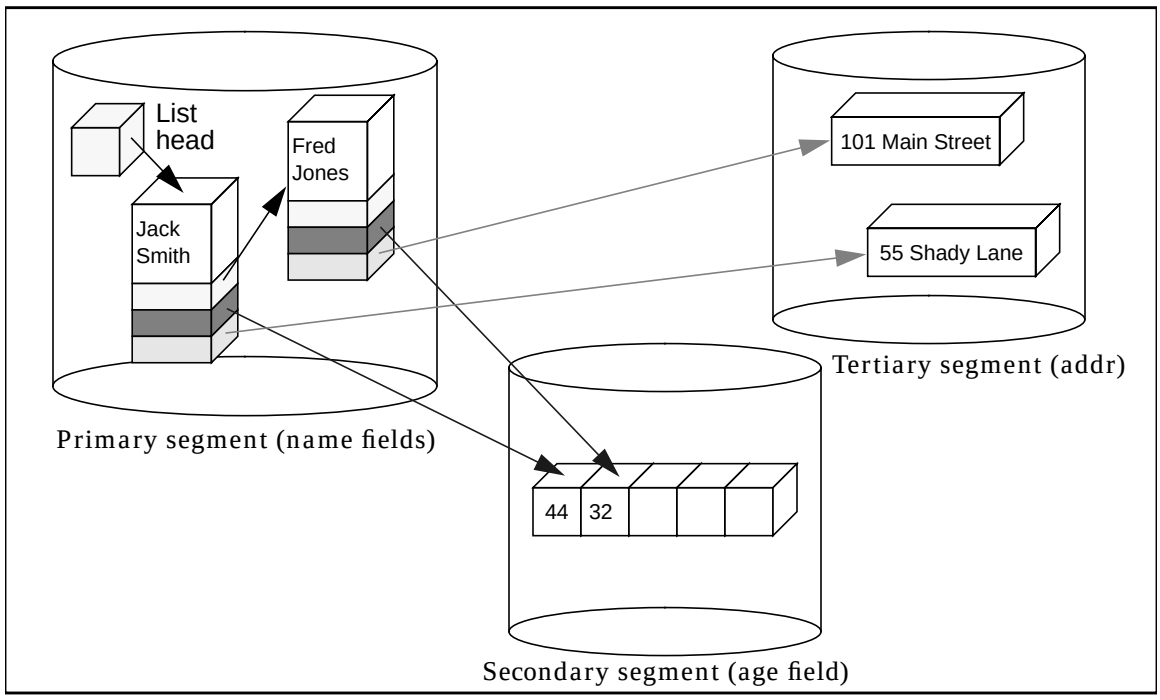

Figure 3.4: A segmentation data structure

Now consider an **INSERT** operation. For each insert of a customer element, three new elements must be created. The first element contains all of the fields in the original customer up to and including **l_name**. The element is dynamically allocated, and placed on a linked list. The second element contains all of the fields from customer from **l_name** up to (and including) **age**. The element is placed in a static array. The final element contains the rest of the fields from customer, and is simply **malloc**'ed. All three new elements must be interlinked (so the compiler can find them later). All of this code must be generated by the compiler, hidden from the programmer's view. In addition, the segment example requires that the compiler perform field and element renaming, to keep the sub-segments properly referenced.

Finally, the ordering of layers in type equations often is significant. Since code generation is performed in the order shown in the type equation (from left to right), cases do exist where changing the order of layers will result in radically different, but predictable, behavior.

3.3.6 Domain-specific optimizations

One of the greatest strengths of NPTs is that they allow for a large number of domain-specific optimizations to be performed. We feel that these optimizations are a key to producing components with competitive performance characteristics. As we mentioned in the introduction, NPT components must perform well, or they will not be used. Our compiler thus contains an NPT optimizer module.

Domain-specific optimizations are possible with NPTs, mainly because of the consistent interface they present. Since the form and content of each component is restricted by the interface, we can make assumptions about components and can perform optimizations on the generated code.

For example, the segment example shown in the previous section affords the compiler a chance to perform an optimization. For each cursor declared on a segmented container, the compiler actually declares and maintains three distinct cursors - one for each of the sub-containers. Now consider a **FOREACH** operation. The code that is generated for a segmented **FOREACH** requires that in each iteration of the **FOREACH** statement's loop that each of the sub-cursors be set to reference the proper element. Consider the following **FOREACH**:

```
FOREACH(seg_curs)
{
  printf("Last name is: %s\n", l_name);
}
```

In this example, each iteration of the **FOREACH** operation only needs to access the contents of the fields in the element in the primary sub-container. The optimizer has knowledge about both the segmentation component and the type equation for the schema, and can thus eliminate the pointer assignments for the other sub-containers (for this **FOREACH** statement only).

In addition, our compiler also assumes that the component writers use a consistent naming scheme for variables in the code generated by the components. This allows the compiler to make assumptions about type information, based on identifier names.[3] Not only does each component export a standardized interface, but the code each component generates (for a given externalized interface) must follow certain rules and conventions set up by the opti-

3. P2 provides the **xp** (eXPand) tool to help generate components from specifications. A tool such as this can be used to standardize identifier names.

mizer. These rules have to do mainly with code structure - to allow the optimizer to combine code fragments from different components in the best possible way. Note that these are not optimizations which would be performed by a language compiler, but rather are optimizations at the higher “code fragment” level.

There are several categories of domain-specific optimizations possible with NPTs:

1. Query optimization allows the compiler to select the component that a cursor will use to traverse a specific container.
2. Layer manipulations are operations performed by the compiler which alter the original type equation provided by the programmer.
3. Traditional code optimizations such as common sub-expression removal, dead-code removal, and useless code removal are performed by the compiler.

In Chapter 4 we will discuss each category of optimization in detail, illustrating with examples from our prototype compiler[4].

3.4 Overcoming TPT limitations

In the following sections, we again list the limitations of TPTs that we introduced in Chapter 2. For each limitation, we briefly describe how the NPT features described above help overcome it.

3.4.1 Difficulty of specialization

We raised the objection in Chapter 2 that TPT authors cannot envision all specializations in advance [Gar92, Sul92]. Even attempting to do so would yield overly large interfaces that would inhibit TPT use. We have observed that the container and link abstractions are remarkably durable. They were recognized in the early 1960's and are omnipresent today. The basic interfaces to these abstractions have been very robust. Despite the robust nature of the interface, the number of generic operations that one needs to perform on containers and links is actually quite small[5].

---

4. Each class of optimization described in this dissertation has been implemented in our prototype. Since Predator is a prototype, we have implemented only a few examples of each type of optimization.

It is crucial, however, that containers be able to have customized interfaces - in some cases radically different from the operations we have provided. This can be performed with NPTs simply by placing a relatively thin veneer on top of a container. The programming language Pascal/R [Sch77], for example, allowed users to customize the interface to relations by letting them place their own ADT interfaces on top of relations and to implement ADT operations as calls to relational operators. All the power of relations (containers) remained, but a customized interface could be used in place of a relational interface.

We, like many others [Boo87], generalize ADT interfaces to generic ADT interfaces. Stacks and queues are examples of generic containers (TPTs) that have special operations (push, dequeue, etc.). These operations can be implemented in terms of container operations (**INSERT**, **DELETE**). Herein lies the key idea for eliminating the difficulty of specializing TPTs. The problem of efficiency for TPT operations is resolved by the compiler and its optimizer, which is responsible for combining the code generated by different components into more efficient source-level constructs.

To place this solution into perspective, Figure 3.5 represents the code of a stack TPT. All implementation details about the stack are inside the cube. Older models of programming expose all of these details to a programmer; specialization is accomplished by recoding the body of the code. This is called the *transparent* model of programming: all details are visible.

At the other extreme, one finds an alternative model of programming where *no* internal details are exposed (TPTs). This effectively corresponds to the availability of TPTs in object (i.e., compiled) form only. As we noted earlier, TPT specialization in this case is not often attempted because of the inefficiencies that ensue. This is classically called the *opaque* model of TPT programming.

Neither the opaque nor the transparent model of TPT programming offers a practical means of TPT specialization. We contend that a *semi-transparent* approach, one that exposes some, but not all, implementation details is a better solution. The details of container implementation are not exposed; only the customized container interface and the mappings of its operations to standard container operations are visible and available for modification. This makes TPT specialization *significantly* easier.

---

5. Evidence to support this statement also comes from network protocols where a simple and standardized interface has worked well for the *x*-Kernel [Pet90, Hut91, OMa90].

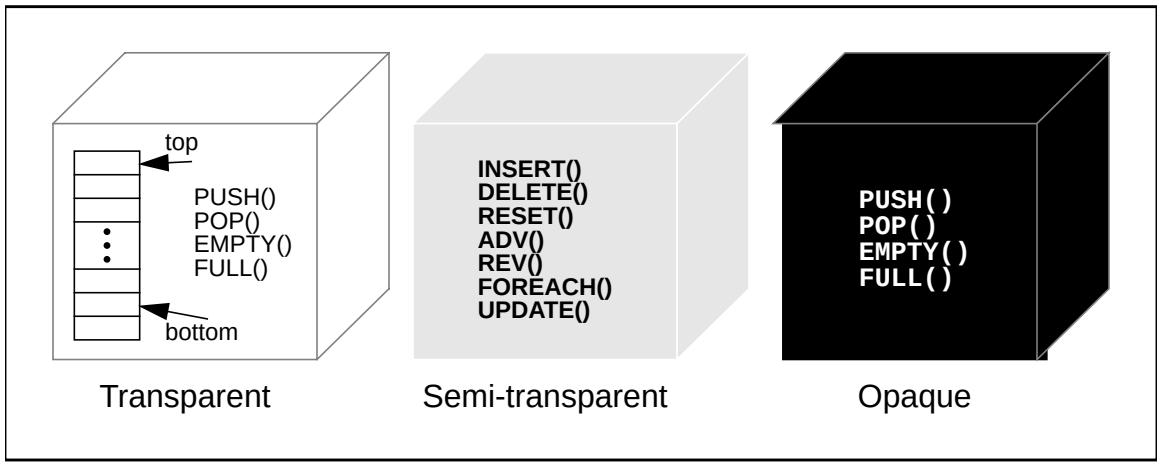

Figure 3.5: A Semi-transparent stack

As an example, a simple (push, pop) interface to a stack container is shown below:

```
MACRO push(container, element)
{
  INSERT(container, element, container.stack_head, AT_END);
};

MACRO pop(container, element)
{
  if (!is_empty(container))
  {
     GETREC(element, container.stack_head));
     DELETE(container.stack_head);
     REVERSE(container.stack_head);
  }
};
```

It has been our experience that the **DS** interface we proposed has been more than adequate to emulate TPT-native functions, as well as applications for which we have desired customized ADT interfaces. An example of this, the supermarket checkout example, is shown in Appendix E.

3.4.2 Complex compositions

Both of the examples shown in Section 2.2.2 are complex compositions which are not easily representable with TPTs. NPTs, however, deal with these compositions in a simple and direct fashion. Compositions of multiple data structures over a single element type are nothing more

than the vertical parameterization described in Section 3.3.3. Our NPT model allows for as many layers as needed to be composed over any base element type.

The other example of a complex composition is a simple instance of a **LINK** realm instantiation. One of the strengths of the NPT approach is that radically different *classes* of layers (such as interlinking elements from different containers) can be classified into a new realm which inherits strengths from the other NPT features. While we have needed only two realms for our work, it is entirely conceivable that further work in the data structure domain might uncover other realms, which we could easily add to our system.

Finally, the complex compositions we have described are possible because of the compiled environment we impose on TPTs. As described previously, simple macro-expansion is not sufficient for some of the compositions we have encountered. Coupled with a domain-specific optimizer, NPTs have considerable expressibility.

3.4.3 Type transformations

Our NPT model is a compiled transformation system, not unlike those of [Coh89, Coh93, Sno89]. Our type equations take an abstract description of a data structure schema and transform it (via our compiler technology) to a concrete realization of the schema. The abstract/concrete paradigm is followed rigorously as the type equation is being resolved. Each layer is responsible for an abstract-to-concrete transformation. The layer is unaware of the implementation details for any of the layers “below” (further to the right) in the type equation. We can map from an abstract to concrete stream of code, for any primitive function, simply by recursively evaluating the type equation, one layer at a time.

Thus, all of the element and container transformations we have described (such as segmentation and compression) are easily encapsulated as NPT layers. Each of these layers performs its mapping, unaware of mappings performed either above or below in the type equation.

3.4.4 Ad hoc interfaces

The consistent, high level interface of NPTs is ideal for dealing with the limitation of ad-hoc interfaces. Since we restrict the interfaces of all layers belonging to a given realm, we can substitute, with no loss of generality, *any* combination of layers in a type equation and recompile

it successfully. That is not to say that the new type equation will be desirable, or even correct; only that it will compile successfully. In most cases, however, simple layer substitution will generate correct working replacements for the original program. The number of legal but meaningless compositions is fairly small.

The small differences in semantics are usually restricted to marginal areas such as bounds and scanning. For example, the **`malloc`** layer will dynamically allocate memory up to the available memory of the machine. If one were to replace the **`malloc`** layer with an **`array`** layer (which is of fixed size), it is possible that an application might receive an "out-of-bounds" error, whereas the **`malloc`** version of the program would continue to function properly. These minor semantic differences are easily tracked and present few problems in practice.

In addition, NPT macros are implemented in terms of the consistent high-level interface. Thus, they too can be recompiled for most data structures with little difficulty. For example, priority queues are often implemented with either arrays or tree-version heaps. But there is no reason that an NPT implementation of a priority queue could not be compiled with an AVL-tree or a hash table. It might not be particularly efficient, but it will compile and execute correctly.

3.4.5 Evolution and maintenance

We have previously stated that a major goal of this research is to allow programmers to write data structure programming faster and more easily than before. This concept extends well beyond the initial writing of an application program. Our NPT model allows application writers to maintain and evolve their systems.

Our NPT model provides several evolution and maintenance benefits. The consistent, high level interface allows programmers to write their programs much closer to the actual algorithmic level. Future modifications of their program are also made at the higher level and are, therefore, simpler. The vertical parameterization of NPTs allows for more complex compositions to be created. This allows programmers to arrive at their desired data structures in fewer iterations. Further, our NPT interface can be specialized via the compiler's macro facility. This allows programmers to add new functionality to existing programs without having to rewrite the data structure (layer) module.

Finally, our model allows for the declaration of both intra-container and inter-container invariants. Cursor selection predicates and predicate indexing layers allow programmers to

declaratively state search conditions on containers, which the compiler maintains automatically. Further, the **LINK** realm layers maintain invariants on the link conditions. They will automatically maintain the declared invariant, and will notify the program (via status codes) of attempts to violate the invariant. This allows application programmers to maintain a higher level of control over their programs as they evolve.

3.4.6 Scalability

A primary goal of this work is to allow the size of our NPT layer library to scale gracefully with the size of the feature set it supports. We feel it is crucial that our NPT design avoid the combinatorics problems presented by libraries such as [Boo87, Lea88, Boo90].

Our NPT layering structure has been specifically designed so that all of the NPT layers are effectively orthogonal to other layers within the same realm. In other words, each new layer that is added to an existing realm is completely independent of any existing layers within that same realm. There is no requirement for any layer to possess knowledge about any other layers, or their interactions. We accomplish this mainly via our compiler strategy. Each layer is responsible for producing the code fragment that implements the requested feature of its layer. It is the responsibility of the compiler to gracefully merge the code fragments from the different layers to construct an overall code fragment that possesses the proper semantics. A discussion of one implementation of this combination strategy (with *functional templates*) is provided in the next chapter. An in-depth discussion of NPT scalability is given in [Bat93a].

3.4.7 Code efficiency

Efficiency is a key concern in this research. It is crucial that our compiler generate efficient data structure code.

The compiler is responsible for generating only the code necessary for the declared operations. Unlike more traditional database systems [Kor91, Bat90, Bat93b], NPT-generated code is *not* general. It is geared specifically for the type equation of a particular container, and incurs only the overhead for the mechanisms to support the primitive functions used in a given application. Thus, a different application which uses the same type equation, but different primitive functions, would not generate the same code. In a similar vein, two copies of the

same application, instantiated with different type equations, would also generate different output code. The key to generation of efficient code is to eliminate excess overhead.

The optimizer is also responsible for improving the efficiency of NPT-generated code. The combination of consistent, high-level interfaces, vertical and horizontal parameterization, and a compiler enables us to perform domain-specific optimizations which are not possible for pre-written (monolithic) or inherited components.

It is possible to write many of these optimizations manually. It is our experience, however, that most of these optimizations are not placed in hand-written code, due to the additional time required to code and debug them. It is our hope that the automatic generation of these optimizations will yield a net efficiency gain over most hand-coded applications.

Chapter 4

Predator: an NPT implementation

We validated our NPT concepts by building a prototype data structure compiler, called Predator. Predator is a two-pass optimizing compiler. It is comprised of approximately 17,000 lines of code, 7,000 of which implement seventeen distinct layers in the **DS** and **LINK** realms. Descriptions of the layers currently implemented are presented in Section 4.2.1 and Section 4.2.2. Predator is written in the C programming language, and has been ported to the following operating systems: Ultrix, SunOS, AIX, OS/2 and DOS.

This chapter discusses several important parts of our Predator implementation. Section 4.1 describes the actual Predator compiler. We discuss how Predator programs are written and compiled, the structure of the compiler, how functional templates and Predator macros are used to generate code, and the types of domain-specific optimizations possible. In Section 4.2 we list and describe the **DS** and **LINK** realm layers we constructed for our prototype. Finally, Section 4.3 details our link cursors and the mechanisms we used in implementing them. Our NPT model specified that link cursors were to exist only as an internal, hidden construct. They turned out to be so useful, however, that we augmented our model and exposed their interface to the application programmer.

4.1 The Predator compiler

Predator programs are C programs which have been augmented with Predator declarations and statements. These programs are called .dac source files (short for data structure C). Predator translates .dac files into C-language source files, include files, and a makefile for

compiling the program. The Predator-generated files are then compiled and linked by any standard C-language compiler. The Predator authoring sequence is shown in Figure 4.1.

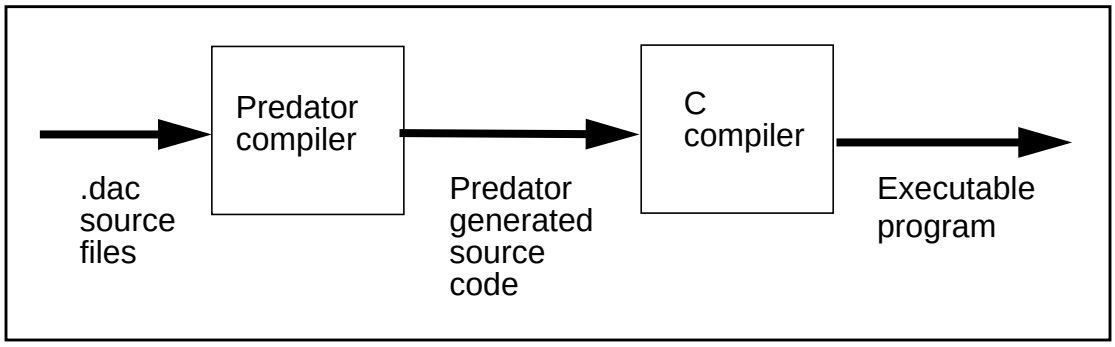

Figure 4.1: Predator authoring sequence

This chapter examines the more significant aspects of the Predator prototype, as well as trade-offs made in our implementation.

4.1.1 Two-pass compilation

As mentioned above, Predator is a two-pass optimizing compiler. Note that Predator differs from standard compilers only in the first-pass optimization stage:

1. **Symbol table creation**. Primitive entities such as elements (structures), schemas, containers, cursors, and links are recorded in the symbol table. As Predator parses the input text, the relevant information about each entity is evaluated and stored. Predator also notifies the programmer of any declared entities that are not used in the source program, and removes them from the symbol table for the second (code generation) pass.

2. **Errors and warnings**. Predator notifies programmers of common problems, such as syntax errors. In addition, Predator issues diagnostic messages when it detects incorrect usage of any primitive function. It notices when incorrect or badly formed arguments are presented, as well as missing arguments.

3. **Optimization**. Predator performs a series of optimizations at the end of the first pass. Mostly these are modifications of the tree representations of type equations that Predator stores in the symbol table. Predator may add, delete, modify, or reorder layers within type equations.

The second pass of Predator performs code generation and code optimization. While the discussion of the detailed approaches and algorithms Predator uses to generate and optimize code would be far too long to include in this dissertation, three important second-pass features will be described: functional templates, recursive macro processing, and code optimizations.

4.1.2 Functional templates

Predator augments the standard C-language with several *primitive functions* such as **INSERT, DELETE, FOREACH**, etc. When Predator encounters a primitive function it must generate a code fragment that implements the function for the associated container and cursor types. The primary method that Predator uses is that of a *functional template.*

A functional template is comprised of code fragments; some are dependent on the type equation (i.e., data structure), while others are not (and are included in a *conceptual* layer, which is used internally by the compiler, but never declared by the application programmer).

As an example, a functional template for the **FOREACH** statement is shown below. Each underlined function generates a code fragment that is type-equation dependent; non-underlined code is type-equation independent.

Predator generates code for primitive functions by traversing the type equation tree for each of the underlined functions, thus generating the code to fill the place holders. The generated code is combined with the non-underlined sections. The overall code fragment is then passed on to the optimizer, and is finally placed on the output code stream. The implementation of templates is accomplished with Predator macros, which are described in Section 4.1.3.

The **FOREACH** function template is:

```
set_to_first_item();
while !(past_end_of_container?())
{
  if ((cursor's predicate) &&
      !(disqualification_predicate()))
  {
     /* Code fragment for FOREACH goes here */
  }
  advance_cursor();
}
```

Underlined functions are called *snippet* functions; these are the functions that are exported (and implemented) by Predator layers. Generally, snippet functions do **NOT** corre-

spond to the set of functions that are exported to Predator users; rather, they are more primitive functions which are needed for code generation. For more information about snippet functions, see Appendix F.

As an example, four snippet functions are referenced in the **FOREACH** template. The first positions the cursor to the first element in the container. The second generates a boolean test to determine if the cursor has finished traversing the container. The third adds any additional tests that layers in the type equation might require to ensure that the code of the **FOREACH** will be executed for the current element. An example of a layer that would contribute code for this snippet function would be the delflag layer (don't execute code for deleted elements). The fourth snippet function is responsible for moving the cursor to the next element of the container.

The layer-based code sections are generated as follows: Predator uses a dispatch table to call on the appropriate snippet function for the layer which is located at the root of the equation tree. This function may or may not generate a code fragment. If a layer is terminal, it returns its generated code fragment up the call chain. If a layer is a non-terminal, it calls on each of its child layers in turn via the dispatch table (most layers only have one child layer), and appends any returned code to the end of its code fragment. The snippet function returns the code fragment it generates (with the appended fragments from lower layers). Correctness of layer-dependent code is dependent on the compiler macro expanding the code based on both the type equation tree and the fact that the **DS** layers specified are orthogonal. Also, the code optimizer must ensure that the correctness of the code is not affected.

Other sections of code (such as the cursor's predicate or the **FOREACH** code body) have their cursor references expanded as necessary and are then placed directly in the template's output buffer. The entire code fragment, once optimized, is placed on the code generator's output stream for further evaluation. Appendix F lists code templates for the major primitive functions of the **DS** realm.

Consider the following declarations:

```
SCHEMA cust_schema ON ELEMENT CUSTOMER = delflag[array[100]];
cust_schema a_cust_cont;
CURSOR cust_curs ON a_cust_cont WHERE age == 30 || age == 44;
```

A Predator user might write a **FOREACH** statement such as:

```
FOREACH(cust_curs)
{
  printf("Name: %s, %s\n", l_name, f_name);
}
```

Predator would expand the **FOREACH** to the following:

```
cust_curs.cur_item = a_cust_cont.elements0;
while (!(cust_curs.cur_item >= a_cust_cont.next_element0))
{
  if ((cust_curs.cur_item->age == 30 ||
       cust_curs.cur_item->age == 44) &&
      !(cust_curs.cur_item->delflag0))
  {
     printf("Name: %s, %s\n", cust_curs.cur_item->l_name,
            cust_curs.cur_item->f_name);
  }
  cust_curs.cur_item++;
}
```

As in the template definition above, the code generated by the layer's snippet functions is underlined.

4.1.3 Recursive macro processing

Predator includes a *recursive macro processor*. Predator macros are not simple text substitutions (as in the C preprocessor), but are conditional text substitutions which are evaluated at compile-time. There are two distinct advantages of Predator macros:

1. Predator macros are evaluated during (not before) the main compile phase. Thus, macro processing (text substitution) may be interleaved with other compilation operations. Therefore, Predator macros may contain references to Predator primitive functions, and visa versa. This allows the compiler to evaluate (and optimize) functions, which may then include additional Predator macros. The macros are evaluated in turn.

   Using a standard macro facility, it would be impossible to code the Predator functions in terms of macros. Being able to do this greatly simplified the writing of both the compiler and the layers.

2. Predator macros are recursive. Thus, it is possible to have a macro reference itself. Note that this must be done using conditions, or the recursion would be infinite. Predator uses recursive macros seldom, but to great effect. For example, **FOREACH** is defined

as a conditional, recursive macro. **FOREACH**, when applied to a composite cursor, can generate additional **FOREACH** macros, which will operate over container cursors. It is up to the conditional code to ensure that the recursion terminates. This differs from the standard C macro facility, which allows no recursion within macros. The conditional tests in Predator macros are based on an optional final argument in the formal parameters of the macro. If the conditional test fails, the macro is not evaluated. Since the evaluation is performed at compile-time, only certain static comparisons can be performed.

Predator templates (see Section 4.1.2) can therefore be implemented as Predator macros. Most of the code which implements Predator layers and function templates utilizes the macro facility, and is simple and high-level. This approach was taken because explicit layering in data structures often introduces unacceptable performance penalties. [Dav92] is but one work which demonstrates that in-lining (macro expansion) can be very beneficial in generating efficient code.

Predator also exports its macro facility. Programmers may write their own macros and use them anywhere in their source text. We used this method to provide data structure independent push, pop, enqueue, and dequeue routines for our supermarket example.

Most of the tokens that the Predator code generator scans will be neither macros nor primitive Predator functions. Rather, they will be the other constructs (including comments) that make up the rest of the source program. Predator simply transmits these tokens. It does not evaluate them, but rather places them directly into the target source file. Only the precompiler directives, primitive functions, and Predator macros are expanded.

4.1.4 Code optimization

In the previous chapter, we briefly discussed the compiler's optimization engine. This section details several of the code optimizations we have implemented. While these optimizations are specific to our compiler (and domain), we believe that the general categories of optimizations, such as layer manipulations and code optimizations (e.g., dead code, useless code), can be performed on other domains.

Layer manipulations. In general, NPT components are implemented exactly as they are described by the type equations written by programmers. However, there are certain times and cases where the NPT compiler will change the type equation specified by the programmer.

These changes involve adding, removing, and reordering layers within the type equation to help generate more efficient code.

Layers are removed by the compiler if it finds that the programmer has not utilized the layer in any function calls. Layers are often added by other layers (such as a link layer, which will append a new **DS** layer, to the type equation, to implement part of the linkage). Finally, layers can be moved within the type equation to improve performance or correctness of the resulting code. An example of this is the segment layer, which will move all non-scanning layers to the primary segment in order to keep the generated code correct and efficient. This modification is unimportant at the abstract (programmer) level, but is crucial at the implementation level. For example, the following type equation:

```
segment[dlist[malloc[]], a_field, size[malloc[]]];
```

places the size layer on the secondary partition. Thus, the **`size`** function would access both the primary and secondary segments, because the **`size`** field is defined in the secondary segment, and the primary segment is accessed on all queries. However, if the **`size`** layer were to be defined on the primary segment, only that (primary) segment would be accessed by the **`size`** function. Thus, the Predator optimizer changes the above type equation to the following:

```
segment[size[dlist[malloc[]]], a_field, malloc[]];
```

While there is no difference (to the programmer) in the above two type equations, the latter will generate more efficient code than the former.

For any given type equation, there may be several layer manipulations that can be applied successfully. In the current Predator implementation, we have only defined manipulations which are orthogonal and can, therefore, be applied in any order. Clearly, there are manipulations which will realize different performance results based on the order in which they are applied. However, we have not concentrated on this issue, and it has been left for future work.

Query optimization. One of the more common optimizations performed by the compiler is query optimization. Most layers provide a mechanism for traversing a data structure of the implemented container. For example, array, linked list, and binary trees implement containers in different ways; each has its own distinct algorithm for container traversal. It is the compiler's responsibility for choosing the layer whose retrieval performance would be most efficient (on average).

Cursor syntax allows for the optional declaration of a **WHERE** clause, which is used to designate the subset of elements visible to the cursor. The compiler uses this clause and a series of metrics[1] (about the query type) to rank each data structure (from most efficient to least efficient) for processing the query. Given a query, it is possible to determine which layer most likely offers the most efficient processing for the query. The compiler adjusts its internal symbol table so that all functions which reference the cursor utilize the proper layer. Consider the following declarations:

```
SCHEMA query_opt ON ELEMENT customer =
       predind[sarray[100, emp_id],
               state == “TX” || state == “WA”];
query_opt a_container;

CURSOR c1 ON a_container WHERE emp_id > 50;
CURSOR c2 ON a_container WHERE state == “TX || state == “WA”;
```

The container in this example has two possible ordering layers, a sorted array, ordered by employee id, and a predicate index (which only indexes records satisfying the supplied predicate). The query optimizer determines that cursor **c1** is best suited for the sorted array, since they are both ordered by **emp_id** (and Predator can use binary search to improve cursor operations on cursor **c1**). On the other hand, the optimizer will select the predicate indexing layer for cursor **c2**, because the predicate for the layer matches the predicate for the cursor exactly.

Sub-expression removal. When snippet functions from multiple layers each produce a code fragment for a primitive function (such as **INSERT**), it is often the case that more than one of them will generate the same fragment of code. In certain cases it is possible to recognize this fact, and remove all but one copy of the fragment from the output source generated. Currently, Predator is able to perform sub-expression removal only on a set of well-known fragments. These are stored internally in the symbol table. While this may seem to be restrictive, remember that the consistent interface of Predator leads to regularly structured layer modules. Hence, the text matching is not as restrictive as it might otherwise be. In future versions of Predator we hope to expand the ability to recognize and remove common sub-expressions.

An example of removing common sub-expressions is setting the cursor status flag. The status flag within a cursor is set at the completion of several of the **DS** realm functions (such

1. Currently, Predator uses ordering (for ordered layers) and textual matching (for predicate indexing layers) as its metrics for query optimization.

as **INSERT**, **ADVANCE**, **RESET**). Most layers can generate a code fragment which sets the status flag. If multiple layers all generate this same fragment, only one of the fragments will be needed. The compiler removes the other fragments from the code generation stream.

Removing dead code. In some cases, snippet functions may produce code fragments which will execute, but do nothing useful. The layer itself does not have this knowledge, but the compiler (being at a higher level) does. In these cases, the compiler can safely remove the fragment(s) of code from the generation stream. For example:

```
SCHEMA dead ON ELEMENT customer =
       segment[array[100],
               name,
               array[100]];
dead a_cont;
CURSOR curs ON a_cont;

FOREACH(curs)
{
  printf("The name field is: %s\n", name);
}
```

where **curs** is a cursor declared on a container of type **dead**, and the **name** field is located in the first (primary) segment. Under normal circumstances, the segment layer generates pointers which traverse each segment during the **FOREACH** function. However, the compiler is able to determine that the only fields of the **customer** element that are referenced in the **FOREACH** (in the **printf** statement) are located in the primary segment. Thus, it discards the (dead) code that would set and advance the pointers for the secondary segment(s). The code before dead code removal is shown below:

```
curs_seg0.cur_item = a_cont_seg0.elements0;
if (curs_seg0.cur_item != NULL)
   curs_seg1.cur_item = curs_seg0.cur_item->seg_1;
while (!(curs_seg0.cur_item >= a_cont_seg0.next_element0))
{
  printf("The name field is: %s\n", curs_seg0.cur_item->name);
  curs_seg0.cur_item++;
  if (!(curs_seg0.cur_item >= a_cont_seg0.next_element0))
     curs_seg1.cur_item = curs_seg0.cur_item->seg_1;
}
```

The underlined code fragments perform no work (since the secondary segment is not referenced in the body of the **FOREACH**), and are removed by the optimizer.

Removing non-executing code. There are cases where a snippet function for a layer can produce code which will not execute. As in the case above, there is no way that the layer itself can know this fact. The compiler is responsible for determining that the fragment will never execute, and removes it from the code stream. An example of this would be as follows:

```
CURSOR curs ON a_container WHERE age > 30;

FIND(curs, age < 30);
```

The **FIND** statement will search for the first element that the **curs** cursor can find whose age is less than 30. The **FIND** function of the scanning layer generates the appropriate code. But the compiler has knowledge that all **FIND** functions are further qualified by the cursor's predicate. Since that predicate invalidates the find (no age can be less than *and* greater than 30), the interior of the body of the **find** can be removed. Note: the entire body of the find cannot be removed in this case. The code for setting the status flags must be maintained.

Predator currently pre-defines a general set of code patterns to be searched for locating code fragments that do not execute. As a result, the optimization is not commonly used. We have hopes, however, that a small rules engine could drive such a mechanism in a more comprehensive manner. Of course, we also hope that programmers do not generally define type equations, cursors, and primitive functions which generate non-executing code.

Bounds checking. Ordered layers offer possibilities for bounds optimizations. Consider the following schema which defines a doubly-linked list, ordered by the age field:

```
SCHEMA bounds ON ELEMENT customer = dlist[malloc[], age];

CURSOR curs ON a_container WHERE age < 30;

FOREACH(curs)
{
  printf("Name: %s\n", name);
}
```

where **a_container** is a container on **bounds**, and **name** and **age** are fields of **customer**. The **FOREACH** statement is responsible for printing the **name** field of each element in **a_container** where the **age** field is less than 30. In this instance, the compiler can notice that the linked list and the cursor selection predicate are ordered on the same field. Additional bounds checking code is added, so that the **FOREACH** will stop when the **age** field of the elements equals or exceeds 30. Again, this is knowledge that the layer itself does not possess. The com-

piler optimizer is required to combine the two pieces of information to process the optimization.

Other optimizations. Other optimizations exist (such as loop unrolling) which we have not investigated which are analogous to classical compiler optimizations. We believe that a general-purpose compiler should be able to handle many of these optimizations. It should be mentioned that the code generated by Predator (and by its optimizer) is not designed to help the optimizer of the language compiler used to compile the Predator output. The Predator output code should be no easier or harder to compile and optimize than any other C-language source.

Optimization interaction. We recognize that there are opportunities for optimizations of one type to be performed based on optimizations of another type. For example, it is possible that reordering the layers within a type equation might cause a code fragment (generated by a reordered layer) to become a piece of useless code and be eligible for removal. The whole issue of interacting optimization features has not been addressed in the current prototype, and has been left for future work.

4.2 Layers

This section describes each of the layers that we have implemented in our prototype compiler. Both **DS** and **LINK** realm layers are discussed. In addition to a general description of the layers, we present a short discussion of some of the more special layers and the difficulties they imposed on the design of Predator.

4.2.1 DS realm layers

The following table lists each of the **DS** realm layers we have implemented. The parameter *ds* is instantiated with an expression of type **DS**.

Table 4.1: **DS** realm layers

Layer	**Description**
ARRAY[*n*]	Unsorted, static array of size *n*.
DELFLAG[*ds*]	Logical deletion layer. Provides a delete-flag field.
MALLOC[]	Forces elements to be dynamically allocated.
DLIST[*ds* (, *fld*)*]	Doubly-linked list. Unsorted, or sorted by ordering specified by optional *fld*(s).

Table 4.1: **DS** realm layers

Layer	**Description**
AVAIL[*ds*]	Avail list for deletion and allocation.
SIZE[*ds*]	Counts the size of the container.
SEGMENT[*ds0*, *fld0*, *ds1* (, *fldn*, *ds(n + 1)*)*]	Partitions the element into multiple data structures. Partition the element at fields *fld0*, *fld1*...*fldn*. Sub-data structures are specified by the *ds0*, *ds1*...*ds(n + 1)* type expressions.
PREDIND[*ds*, *pred*]	Predicate-based indexing. Elements are added to index if they satisfy predicate *pred*.
BINTREE[*ds*, (*fld*)+]	Binary tree ordered by field(s) *fld*+.
ACTION[*ds*, *fctn*, *hint*, *code*]	Provides before and after actions for functions. Perform code fragment *code* for primitive function *fctn*. *hint* is either **BEFORE** or **AFTER**.
USAGE[*ds*]	Allows MRU and LRU for container access.
PINDEX[*ds*, *pred*]	Another implementation of predicate indexes.
TIMEST[*ds*, *hint*]	Global or local time stamps for each element. *hint* determines global or local timestamp.
PERSIST[*file*, *size*]	Elements are stored persistently in file *file* whose maximum size is *size*.
SARRAY[size, (*fld*)+]	Sorted, statically declared, fix-sized array of size *size*. *fld* determines the ordering fields.

There are four terminal layers currently defined in Predator. **ARRAY** causes elements to be stored in an unsorted, statically declared array. The size of the array is a parameter. **SARRAY** is similar to array, only the elements are sorted based on the fields specified in the type equation. **MALLOC** causes elements to be allocated and freed dynamically from a heap. Finally, **PERSIST** allows elements to exist in a memory-mapped persistent store. The file name and size of the store are parameters of **PERSIST**.

AVAIL is not a terminal, but is tightly coupled with the terminal layers. It provides an avail-list of previously deleted elements. When an element is deleted from the container it is not destroyed, but simply placed on the avail-list. When the next **INSERT** call is made, an element is taken from the avail-list (if one is present). This layer is useful because it is often faster to reclaim previously allocated space than to use the allocation algorithm of the native terminal layer (such as **MALLOC** or **PERSIST**), and because it helps reclaim space from the interior of arrays (Predator arrays do not provide element reclamation).

Some layers provide basic ordering and indexing operations. **DLIST** declares a sorted (or unsorted) doubly linked list. **BINTREE** provides an ordered binary tree. There are currently two predicate indexing layers: **PREDIND** and **PINDEX** are very similar, differing mainly in their internal representation. The former utilizes data fields within the elements to store pointers of the predicate index, whereas the latter stores pointers of the predicate index outside of the elements. Both are illustrated below in Figure 4.2 for an index which links elements where: **(age == 5) || (age == 3).**

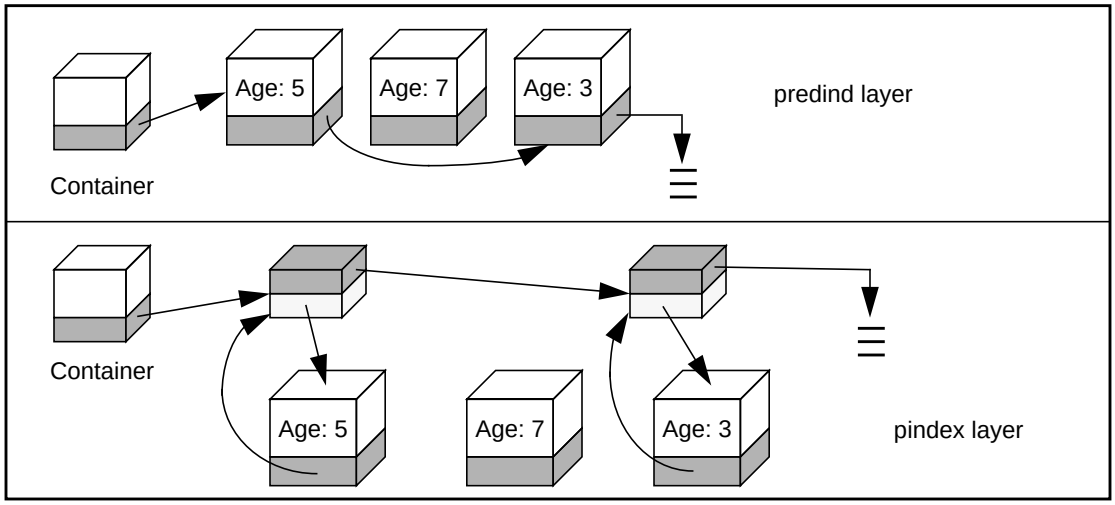

Figure 4.2: Two approaches to a predicate-based index

Several of the layers export their own unique functions. The **SIZE** layer is responsible for keeping track of the size of a container and exports the *size* function to do that. The **USAGE** layer keeps track of the most and least recently used elements in a container. It exports the functions **MRU** and **LRU**. The **TIMEST** layer maintains either local (to the container) or global (to the system) timestamps for each item. Timestamps are automatically updated on **INSERT**, **DELETE**, and **UPDATE** operations. The layer exports the **TIMEST** function.

The **ACTION** layer allows programmers to specify special code fragments which are performed either *before* or *after* a specified primitive function. For example:

```
action[malloc[], INSERT, AFTER, printf("Hello!\n")];
```

would cause the printf statement to be placed after each **INSERT** function called in a program. However, this will only be done for those **INSERT** statements which take place on containers whose type equations contain the action layer. All other **INSERT** statements are not affected.

DELFLAG is a layer that encapsulates logical deletion. Each element is augmented with a boolean variable to denote whether or not it has been deleted. Insertion clears this flag, while deletion sets it. Cursor retrievals examine the flag to ensure that deleted elements are not returned to users.

The **SEGMENT** layer encapsulates a form of element fragmentation. Each element of the container is partitioned into two (or more) new elements, where corresponding sub-elements are stored in their own separate container. These new sub-containers are also created by the segment layer. Segment, therefore maps between the unfragmented view of elements (visible to the application) and the fragmented implementation. While segment is transparent to the application programmer, it can have significant performance effects. An example of this is presented in Chapter 5.

4.2.2 LINK realm layers

Links are declarative constructs in Predator. Links connect two containers together within a link relationship. It is important to remember that this relationship is between *containers*, not schemas. It is possible to declare multiple containers for a given schema, only one of which participates in a link.

The following table lists the implemented layers in the **LINK** realm:

Table 4.2: **LINK** realm layers

Layer	**Description**
LINKNEST[*lnk*]	Nested-loop link algorithm.
LINKPOINT[*lnk*]	Pointer-based link algorithm.

In the table above, the argument *lnk* represents either a **LINK** or **DS** realm layer. Thus, type equations can be adorned with any number of **LINK** layers followed by only **DS** layers (since no **DS** realm layer is parameterized in terms of **LINK** layers). Thus, **LINK** can be thought of as a super-realm of **DS**, implementing the entire **DS** interface, as well as those functions shown in Table 4.3 and Table 4.4.

Currently there are only two link layers implemented. Nested loop linking (**LINKNEST**) is very simple. No actions are taken by the link layer for inserts, deletes, or updates. For each cursor traversal, a cross-product of the two containers is performed (via a **FOREACH** statement

for each container), with qualification being determined by applying the link predicate to each pair of cursors.

Pointer-based links (such as **LINKPOINT**) are more complicated. The **LINKPOINT** layer is responsible for maintaining physical linkages between elements that are logically related in different containers. Figure 4.3 shows a **LINKPOINT** implementation between department objects and the employees in those departments. Note that each element in each container is connected to a list of pointers to the elements in the other container to which it is logically related. Moreover, each container is adorned with a header field to another linked list, which identifies elements which participate in the join (This, in effect, implements a semijoin [Kor91] of that container with the other container). Figure 4.3 depicts a populated link with the following declaration (the numbers in each element are the **dept_num**):

```
LINK one dept TO many employee USING linkpoint
        WHERE dept.dept_num == employee.dept_num;
```

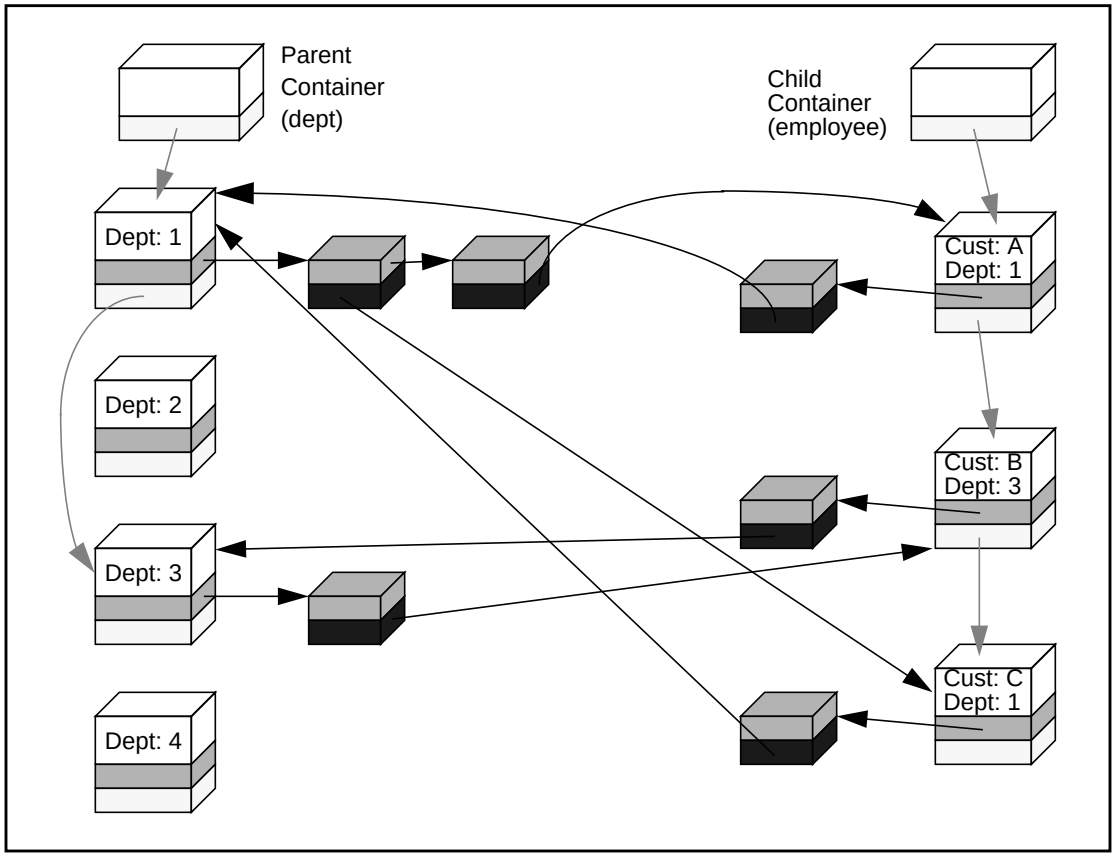

Figure 4.3: A pointer-based linkage

While links are discrete entities on the abstract level, Predator actually implements links as part of the **DS** realm containers. When a link is declared in a Predator program, the compiler places a layer in the type equation of both the parent and child containers. Remember that type equations are based on schemas (not containers), and there can be more than one container which shares a particular schema declaration.

Since a given link only applies to one pair of containers, Predator must ensure that the link only applies to the containers of the link, and not to any other containers which share the type equations of the containers in the link. Predator achieves this by adorning the link layer in the type equation with a text field that identifies the container for that link. The link layer added to the type equation will only be traversed for operations for the specific container of the link. Consider the following declarations:

```
SCHEMA cust_schema ON ELEMENT customer = dlist[malloc[]];
SCHEMA dept_schema ON ELEMENT customer = array[1000];

cust_schema cust1, cust2;
dept_schema dept1;

LINK a_link ON ONE dept1 TO MANY cust1 USING linkpoint
     WHERE cust.dept_num == dept.dept_num;
```

In this example, both container **cust1** and container **cust2** share the same type equation. Predator adds additional layers to the type equation for container **cust1** for the purposes of code generation for link **a_link**. Since container **cust2** does not participate in **a_link**, it is important that the added layers be able to be differentiated, so that the layers are used when the type equation is traversed for **cust1**, but not when traversing the type equation for **cust2**.

Thus, Predator modifies the two type equations above into these new type equations:

```
SCHEMA cust_schema ON ELEMENT customer =
       linkpoint[dlist[malloc[]], cust1];
SCHEMA dept_schema ON ELEMENT customer =
       linkpoint[array[1000], dept1];
```

The link layers implement a superset of the **DS** realm snippet functions. This is necessary because the link layers not only have to support **LINK** realm functions, but may also have to generate code fragments for such **DS** realm operations as **INSERT** and **DELETE**. For example, each time a container linked via linkpoint has an **INSERT** operation performed on it, the linkpoint layer must add a code fragment which determines if the new element should be linked

with any existing elements in the other container. If so, the code fragment must add all of the proper linkages to elements in both the parent and child containers. The size of this code fragment can be large, but Predator hides the complexity from the application programmer.

4.3 Link cursors

In Chapter 3, we described the **LINK** realm and its interface. In our Predator prototype, we chose to implement the abstraction of link traversal with a *link cursor*. Link cursors have been a very useful construct because their simplified interface (coupled with the recursive macro facility) greatly eased the task of writing link layers. Link cursors have been so useful that we have exported them to the application programmer.

Link cursors are declared as follows:

```
LCURSOR <lcursor_name> ON <link_name>
                       USING <parent_curs>, <child_curs>;
```

where **<lcursor_name>** is the name of the link cursor, **<link_name>** is the name of a declared link, and **<parent_curs>** and **<child_curs>** are the names of two cursors declared on the parent and child containers for the link. Multiple link cursors may be declared on a single link, and there is no limit to the number of link cursors that may share a particular parent or child container cursor.

A link cursor is an abstraction for a runtime object that traverses a link. At any given time, the link cursor references a *pair* of elements, one in the parent container of the link, and one in the child container (where the parent and child containers are usually different containers). Advancing a link cursor causes it to reference the next pair of elements that satisfies the link's predicate. The end result of using a link cursor is that the two constituent container cursors are set to the proper elements in the containers (independently of the **LINK** layer used).

Programmers reference link cursors either through the container cursors or through the link cursor itself. For example,

```
LCURSOR my_lc ON my_link USING par_curs, child_curs;
```

declares a link cursor on the link *my_link*. Suppose that *my_lc* is later set to point to a valid pair of elements in the link. The programmer could refer to a field in the parent element with *any* of the following expressions:

```
par_curs.fld;
my_lc.parent.fld;
my_lc.par_curs.fld;
my_lc.fld;              /* This assumes that fld is defined only
                           in the parent container            */
```

Link cursors are used in primitive functions in the application code, much the same way as in container cursors. The following table lists the syntax available for link cursors:

Table 4.3: Link cursor functions

Function call	**Meaning**
RESETL(lc)	Reset link cursor *lc* to the first pair of elements that satisfies the link.
ADVL(lc)	Advance link cursor *lc* to the next pair of elements that satisfies the link.
ENDL(lc)	Test to see if link cursor *lc* is past the end of the link, or references a valid link element.
FOREACHL(lc) {code}	Execute code fragment *code* for each pair of container cursors referenced by link cursor *lc*.

Table 4.4 describes a set of primitive functions which are used (internally) to implement the functions of Table 4.3. However, they also allow programmers to access link cursors in another way. Being able to bind one of the two container cursors of a link cursor (outside of link cursor operations) can be useful in processing complex queries. We will see examples of this later. For now, we note that Predator offers primitive functions that operate on only one cursor of a link cursor (while assuming that the other cursor's position has already been fixed). There two sets of such functions; one set operates on unbound parent cursors (assuming a bound child cursor), while the other operates on unbound child cursors (assuming a bound parent cursor). Corresponding function names in each set have the same name, except that the final character is either **p** or **c** (for parent or child). Table 4.4 summarizes these functions.

Table 4.4: Parent/child functions for link cursors

Function call	**Meaning**
FIRSTC(lc)	Position the child cursor to the first child for the given parent.
LASTC(lc)	Position the child cursor to the last child for the given parent.
NEXTC(lc)	Position the child cursor to the next child for the given parent.

Table 4.4: Parent/child functions for link cursors

Function call	**Meaning**
PREVC(lc)	Position the child cursor to the previous child for the given parent.
ENDC(lc)	Test to see if the child cursor is past the last child for the given parent.
FOREACHC(lc) {code}	Execute code fragment *code* for each child of the given parent.
FIRSTP(lc)	Position the parent cursor to the first parent for the given child.
LASTP(lc)	Position the parent cursor to the last parent for the given child.
NEXTP(lc)	Position the parent cursor to the next parent for the given child.
PREVP(lc)	Position the parent cursor to the previous parent for the given child.
ENDP(lc)	Test to see if the parent cursor is past the last parent for the given child.
FOREACHP(lc) {code}	Execute code fragment *code* for each parent of the given child.

The following is an example utilizing these functions:

```
LCURSOR lm ON manages USING pc, cc;

FOREACHL(lm)
{
  printf("Two department numbers are: %d %d\n",
           pc.dept_num, cc.dept_num);
}
```

This **FOREACH** statement would be expanded to:

```
FOREACH(pc)
{
  RESETC(lm);
  while (!ENDC(lm))
  {
    printf("Two department numbers are: %d %d\n",
           pc.dept_num, cc.dept_num);
    NEXTC(lm);
  }
}
```

This example iterates through all pairs of the link, executing the printf statement for each pair of elements that satisfy the link. Predator generates code first to iterate through all elements of the parent container, via a **FOREACH(pc)** statement. Then (with the parent cursor bound), each child for that parent is examined. This is performed by resetting the child cursor

to the first child (for that parent) with the **RESETC(lm)** function. Then, while the last child has not been found, the **printf** statement is executed, and the next child is visited with the **NEXTC(lm)** statement.

In fact, the code fragment above is actually the code template for the **FOREACHL()** function. When Predator evaluates a **FOREACHL()**, the code section above is instantiated, expanded, and placed in the output code stream. This is an example of the recursive macro facility described in Section 4.1.3.

4.3.1 Using link cursors to emulate composite cursors

In Chapter 3 we defined composite cursors. This section briefly describes the method that Predator uses (during the code-generation pass) to resolve complex composite cursors into the source-level code that implements them.

For each composite cursor **FOREACH** statement that is parsed, Predator constructs a code fragment, comprised of link cursor functions, parent/child link cursor functions, and container cursor functions, which implements the composite cursor. This code fragment is passed back to the scanner/tokenizer/parser for re-evaluation.

A short example illustrates the process:

```
LINK lnk1 ON many cont1 TO many cont2 USING linknest
          WHERE cont1.fld1 == cont2.fld2;
LINK lnk2 ON many cont2 TO many cont3 USING linkpoint
          WHERE cont2.fld3 > cont3.fld4;
LINK lnk3 ON many cont4 TO many cont5 USING linknest
          WHERE (cont4.age == cont5.age) ||
                (cont4.id == cont5.id);

COMPCURS my_cc USING c1 cont1 c2 cont2 c3 cont3 c4 cont4
         c5 cont5 WHERE lnk1 && lnk2 && lnk3 &&
         c1.fld1 > 4 && c3.fld4 < 6;
```

where the containers and container cursors have already been declared. Predator generates implicit link cursors for each link clause listed in the composite cursor. For example, **lnk1_lc** would be used to traverse **lnk1**. These implicit link cursors are used for code generation.

Predator generates a code fragment as follows[2]: A table (**T1**) is created which stores one entry for each conjunctive clause of the **WHERE** clause of the composite cursor. Another table

(**T2**) is created with one entry for each container cursor of the composite cursor. The composite cursor shown above would have the following tables:

Table 4.5: Table (T1) of clauses for a composite cursor

Clause	Link Clause?	Parent Cursor	Child Cursor	Already Bound?
lnk1	Yes	c1	c2	No
lnk2	Yes	c2	c3	No
lnk3	Yes	c4	c5	No
c1.fld1 > 4	No	c1		No
c3.fld4 < 6	No	c3		No

Table 4.6: Table (T2) of cursors for a composite cursor

Container Cursor	Already Bound?
c1	No
c2	No
c3	No
c4	No
c5	No

The following algorithm is used to generate the code fragment:

```
1. Order clauses in T1 so all link clauses precede all container
     cursor clauses.
2. Mark all entries in T1 and T2 as not bound.
3. For each link clause in T1:
   A. If parent and child cursors are not bound (in T2),
        generate a FOREACHL statement, else
      If parent is bound (in T2), and child is not,
        generate a FOREACHC statement, else
      If child is bound (in T2), and parent is not,
        generate a FOREACHP statement,
      If parent and child cursors are both bound (in T2),
        generate a conditional test based on clause.
   B. Mark clause as bound in T1. Mark parent and child cursors for
        the clause as bound in T2.
```

2. Ordering of clauses in the code fragment (from the composite cursor) is based on many factors, such as container and link implementation, hints, and physical order in which the links are declared.

```
      C. For each non-link clause in T1:
         1. If the clause references either the parent or child cursor
              of the link clause, and the clause is marked as not
              bound:
            a. Generate a conditional test based on the clause.
            b. Mark non-link clause as bound.
   4. For each non-link clause in T1 which is marked as not bound
      (i.e. a clause which references a container cursor which is not
      referenced in any link clause):
      A. Generate a FOREACH for the cursor.
      B. Mark the clause (in T1) as bound.
      C. Mark the cursor (in T2) as bound.
   5. Generate the code fragment for the body of the FOREACH.
```

Consider the algorithm for the example above. In this example, Predator first generates a **FOREACHL** statement for **lnk1** using the link cursor **lnk1_lc**, because neither the parent nor the child cursor have been previously bound. Next, Predator determines that one non-link clause in **T1** references one of the two cursors of **lnk1**. A conditional test is generated. Since no other non-link clauses reference **C1** or **C2**, another link clause, **lnk2,** is investigated. Since the parent of **lnk2** (**c2**) is already bound, a **FOREACHC** statement is generated. Again, one of the non-link clauses references a cursor of the current link (**c3**), and a conditional test is generated. The next link clause to be investigated, **lnk3**, references two previously unbound cursors, **c4** and **c5**. Predator generates a **FOREACHL** statement for the clause. Since no other link clauses exist in table **T1**, step 3 terminates. Since all of the non-link clauses in table **T1** have been bound at this point, step 4 does no work. Finally, Predator inserts the code which makes up the body of the **FOREACH** statement on the composite cursor.

The code generated for the example **FOREACH(my_cc)** would be:

```
FOREACHL(lnk1_lc)
{
  if (c1.fld1 > 4)
  {
     FOREACHC(lnk2_lc)
     {
       if (c3.fld4 < 6)
       {
          FOREACHL(lnk3_lc)
          {
            /* Code here */
          }
       }
  }
}
```

Note that there are many possible resultant code fragments for the sample composite cursor; the one shown is just one example.

Composite cursor generation is a good illustration of the evaluative nature of the Predator code generator. A **FOREACH()** statement on a composite cursor requires a minimum of four levels of expansion/optimization to generate the eventual output source code. Those steps are shown below:

1. The **FOREACH(my_cc)** statement is decomposed into statements on link cursors, parent/children of link cursors, and container cursors, such as **FOREACHL()**, **FOREACHC()**, and **FOREACH()**.
2. The link cursor statements are decomposed into statements operating on container cursors, usually **FOREACH()** statements.
3. The **FOREACH()** statements are then expanded into code fragments, including element field references of the form *cursor.field*.
4. The *cursor.field* references are expanded into proper code expressions, based on the composition of layers that make up the containers.

This is by no means the largest number of levels of expansion that may be performed. Rather, it is a minimum. Data structure-specific optimizations may be performed on any of the code fragments at any level of decomposition (based on the composition of layers which make up the containers and links), which may further alter the process. Finally, it should be noted that the entire expression is not evaluated at one time on each level. A lazy algorithm is used, and only small amounts of code are evaluated at one time. Thus it is possible to have code fragments at various stages of completion on the evaluation stack at any time.

4.4 Conclusions

Predator is a result of applying design and implementation decisions to our NPT model. This chapter has described the overall Predator system, the layers we implemented, and several of the more interesting implementation decisions we made. Many of the choices we made were not the best possible; in Chapter 7 we discuss several of the choices, their ramifications, and possible solutions.

Despite these choices, the prototype and layers we constructed have been sufficient to demonstrate our primary thesis. The validation experiments we ran on our prototype, as well as the results we obtained, are detailed in the next chapter.

Chapter 5

Validation

The implementation of NPTs in the Predator compiler has been incrementally validated as the compiler has been developed. As we stated in the introduction, the goals of our work were to provide a domain-specific solution for data structures which would:

1. Allow programmers to code data structures faster and more easily than by hand,
2. Allow programmers to utilize efficient data structures, even if they do not fully understand how those data structures work,
3. Enable programmers to debug Predator programs without having to see or interact with the code generated by Predator (i.e. debug at the algorithmic level),
4. Allow programmers to change the underlying data structure in a program without having to modify the application code,
5. Generate code that is at least as efficient (in execution time) as that which could be generated by hand by the "average" programmer.
6. Aid the programmer in the evolution process by enabling the addition of new features and implementations without having to rewrite the existing code base.

With this in mind, a series of experiments to validate Predator were designed and implemented. These experiments were conducted at different stages of the research, with user productivity experiments being performed early on, and more complicated (systems level) experiments coming later.

The validation work for this research was partitioned into four primary areas:

1. User productivity in writing data structure programming.

2. Comparison of Predator's performance against a commonly used and highly-tuned data-structure component.
3. A view of how the abstractions of NPTs and the Predator code compared against other available solutions for the data structure domain.
4. Use of the Predator compiler to rewrite an existing large-scale application.

5.1 Programmer productivity

A series of experiments were conducted to gauge how a group of programmers would adapt to programming in an NPT environment. Specifically of interest were three questions:

1. How much time would it take to write a working solution to a problem in both Predator augmented C and native C?
2. What would be the efficiency of the resulting programs?
3. How would the above two results change when the data structure changed?

Four different programming problems were created. The tasks varied from very simple (adding, deleting and printing customers using an array) to fairly complicated (maintaining join relationships between an ordered binary tree and a linked list). All of the experiments involved tasks which are fairly common in data structure programming.

Many of the experiment's subjects were professional programmers. Of the total sample of 18 programmers who participated in these experiments, eight of them were full-time C programmers at IBM Austin. The rest were undergraduate or graduate students in the Computer Science department at UT Austin. A full explanation of the experiments can be found in Appendix A.

In each experiment, the subjects were asked to write a C-language version of the program, a Predator version, and an optimized version of the original C code. For comparison purposes we were most interested in the programmer's first working solution. This is because a first working solution to a problem is often the last, due to time constraints.

Only a few of the subjects involved wrote Predator versions of their solutions. Of those who did, half of the subjects wrote the Predator version first, and visa versa. No advantage was found in writing either version first. Each of the programmers who used Predator were

allowed to “play” with the compiler first, to get acquainted with Predator’s syntactic additions to the C programming language.

Each of the experiments measured both the time it took programmers to write their solutions (programmer productivity), and the speed of the resulting program (program efficiency). All programs were compiled with the *gcc* compiler (version 2.4.5), compiled with optimization (-O2), and results compiled using *prof*. All program executions were performed on a DecStation 5000/240 running with 32 megabytes of main memory.

For each experiment, the programmers were provided with the data structure element definitions and all input and output routines. This was done for two reasons:

1. All of the solutions to the experiments looked the same, and could use the same input data file. This made for easier data analysis.
2. The measured differences in performance came only from the data structure specific code added by the programmers.

5.1.1 Coding a simple data structure

The first experiment involved asking the programmer to add, delete, and traverse the most elementary of all data structures - the array. The programmers were given a basic C structure and a static array of data. They were asked to declare a new array and to insert eight different data records, then to delete several of the records (by marking a deletion bit in the record) based on the contents of the *age* field. Finally, they were to traverse the array and print out the remaining (non-deleted) names. While inherently relational in nature, the required subtasks of this experiment are common for data structure programming.

The results of this experiment were quite surprising. With both a simple data structure and simple task to perform, we expected both the programmer productivity and the speed of the resulting programs to be fairly similar for the programmer’s first solution and the Predator version. The table below shows that this was far from the results we obtained[1].

The Predator versions of the program ran, on average, 5% faster than the programmer’s first approach, and about the same as the “optimized” versions. (One programmer’s solution averaged 7.4 seconds, 3% faster than the Predator average). Even for this simple program, it

1. The execution time was arrived at by running the basic loop of the program 10,000 times.

is possible to write more efficient code than that generated by Predator; but, in general, Predator generated code was quite competitive.

Table 5.1: Results of Array experiment

Programming system	Number of subjects	Average time to write (min)	Average execution time (sec)
Hand code (1st pass)	18	22.2 ± 7.7	8.0 ± .3
Hand code (optimized)	9	34.9 ± 9.2[a]	7.7 ± .2
Predator	9	8.0 ± 2.6	7.6 ± .1

a. The time shown (34.9 min) is the average total time for the optimized version, including the time required to code the first pass.

Even more surprising was the observed improvement in programmer productivity. When Predator was first envisioned, it was assumed that Predator would yield a dramatic increase in productivity for complex data structures. What this experiment shows, however, is that even for simple tasks on simple data structures, Predator improves productivity. All of the subjects were experienced C programmers, and all of them had written array data structures many times. Nevertheless, to write the sample task with arrays was quicker in Predator. The programmers reported that most of the gain seemed to be due to the fact that they did not need to worry about conditional tests for deletion and ranges in their Predator versions.

An interesting side-note to this experiment has to do with the uniformity of the Predator solutions. Since the syntax of Predator is fairly restrictive, there are fewer ways to write many common syntactic constructs. For example, the basic type equation in all of the Predator solutions was:

```
SCHEMA cust_ds ON ELEMENT customer = delflag[array[10]];
```

Another example: the programmers were asked to iterate through the array and print out and delete all names whose age is greater than 40, and who were not already deleted from the array. In Predator, the following expression is so simple that it is hard to see how this loop

```
FOREACH(older)
{
  printf("%s\n", name);
  DEL(older);
}
```

could be written in many other ways. As a result, most of the Predator programs were identical (save the identifier names). This leads to the interesting supposition that it may be easier to understand and reuse Predator programs written by others, since the high-level constructs yield more standardized programming. This, however, cannot be tested until there is a sufficiently large community of Predator users.

5.1.2 Programming more complex data structures

While programmers often use trivial data structures (such as arrays) in their programs, they also commonly use more complicated data structures.

Three other experiments, involving queues, multiple index-lists, and interlinked data structures were designed and given to the subjects. Descriptions of these experiments can be found in Appendix A. Each of the experiments was designed to simulate an interesting facet of data structure programming. The experiments were:

1. Queue experiment: The programmers were given a presupplied component library (for a queue), whose interface was not sufficient to program all parts of a given task.
2. Multiple list experiment: The programmers were required to create a data structure which needed multiple copies of the same list indexing structure. They were faced with the dilemma of repeating code or of parameterizing (and generalizing) their data module.
3. Interlinked experiment: The programmers were requested to maintain links among different data structures for easy and quick traversal.

Despite the fact that many programmers undertook each of the three experiments listed above, very few completed them. None of the programmers finished the final experiment (our reference solutions took 130 and 21 minutes for the C and Predator versions respectively). The results for those who did finish are summarized below.

While the completion rate of the experiments was disappointing, it did reinforce the primary motivations behind these experiments. Programming complex data structures by hand takes too long. Ten of the subject programmers began hand-coding one or more of these experiments, only to stop after an average of three hours per experiment. When combined with the results of those who did complete the experiments, it suggests that Predator does indeed offer

the potential of significant productivity benefits. This will be observed more later in this chapter.

Table 5.2: Results of the queue experiment

Programming system	Number of subjects	Average time to write (min)	Average execution time (sec)
Hand code	2	135	18.2
Predator	3	10	15.8

Table 5.3: Results of multiple list experiment

Programming system	Number of subjects	Average time to write (min)	Average execution time (sec)
Hand code	2	210	16.4
Predator	3	11	9.9

5.1.3 Productivity conclusions

The four productivity experiments show that, given experience with both programming systems, Predator shows promise of increased programmer productivity over hand-coding the equivalent data structures. And, as the data structures become more and more complex, the gains should increase. This is due to the fact that the Predator syntax needed to describe complex data structures is no more complex than that for simple structures.

The experiments also showed another strength of the Predator approach. Before each subject was sent a copy of the experiment descriptions, they were asked to describe their knowledge of various data structures. While most of the respondents knew the details of linked lists, binary trees and relational links, some did not. And those who did not were able to program those data structures (in Predator) as quickly and easily as those who did understand the inner workings of those data structures. This is a major benefit of both TPTs and object orientation, which Predator shares.

Finally, an unexpected result was observed. It was shown (primarily with the array experiment) that the efficiency of code generated by Predator often is superior to that generated by hand coding. This will be further explored in the following sections.

5.2 Predator quicksort

While the productivity experiments are interesting, one could claim that the scenarios for the experiments are somewhat artificial. While it is true that the scope of these experiments was limited, they do include many of the basic functions performed on all basic data structures.

The next experiment was designed to address the following concern: How can it be shown that Predator-generated code compares favorably with component-level code that is commonly used, and highly hand-optimized? This is a crucial question. If Predator can not generate code that compares favorably with code used by programmers every day, this work will be of academic interest only.

With this in mind, it was decided to select a code sample and recode it with Predator. We selected the quicksort routine from the BSD version of UNIX for several reasons:

1. It is common used, and is a trusted UNIX utility.
2. Sorting is well understood.
3. It is highly optimized.

Amplifying point 3, we note that the BSD quicksort routine has been refined over many versions such that it highly tuned. In fact, the BSD version of quicksort has been tuned so much that the function operates down to both the byte and bit level - thus making the function quite difficult to read and understand. This will be expanded on later.

We attempted to convert the BSD quicksort routine directly into Predator syntax. This approach failed for multiple reasons. The largest problem with the code port was that the BSD version was written to be quite specific for one data structure. Unlike the eventual Predator version, the BSD quicksort can only sort one data structure - a fixed size array. Since Predator programs are written to be data-structure independent, the conversion was just too difficult. Also, NPTs are designed so that the resultant code will mirror the underlying algorithm as closely as possible, and the BSD quicksort is very *unlike* the standard quicksort algorithm, due to the byte-level manipulations they include for performance.

The next attempt at providing a Predator-based quicksort was far more successful. Since the algorithm for quicksort is common and well understood, we took an existing version of the algorithm from [Aho83] as the base. The conversion of the algorithm to Predator code took less than 30 minutes.

The converted version, while a working quicksort algorithm, was not yet ready to compare against the BSD version. This was because:

1. The pivot selection algorithm was different in the two versions of quicksort. We had to modify the AHU version to match the BSD pivot selection algorithm.

2. The base-case selection for the two implementations was also different. The Predator version was changed to perform a bubble sort for sub-cases of size 5 or less, to match the BSD version.

3. The AHU algorithm contains an optimization that the BSD version does not. Specifically, it notes the special case that all elements of a sub-case have the same value. In this case, no additional sorting or recursion need be done. If there are a lot of duplicates in a given test case, there can be considerable time savings. The BSD version contains no such optimizations. None of our data included duplicates, to ensure that we did not utilize this optimization.

All of these coding changes were made to the Predator version to make it as similar to the BSD version as possible. When completed, the size of the programs were as follows:

Table 5.4: Source code size comparison

Source file	**Number of words**
BSD quicksort	460
Predator source	323
Array (generated)	462
Segment (generated)	531

The table above demonstrates some interesting behavior of Predator programs. The actual source of the Predator version is 70% of the size (in words)[2] of the BSD version. Taken as a rough metric of complexity, the Predator version is smaller and simpler than the equivalent “optimized” BSD version. Complete copies of the two source files can be found in Appendix C.

2. Words were used as a rough measure of complexity, since bytes can be influenced by long symbol names, and lines counts are affected by the number of words per line.

5.2.1 Comparing Predator versus BSD

Tests were then run to compare the efficiency of the two versions of quicksort. A sample driver was written, which did nothing more than place data items in an array, randomize all elements, call on the appropriate quicksort (BSD or Predator), and output the sorted array to a file. The data records that were sorted were C structures with the following declaration:

```
struct customer
{
  char l_name[22];
  char f_name[22];
  int age;
}
```

The size of the name fields of the customer structure were chosen carefully. According to the documentation of BSD quicksort, it is optimized for a structure size of 48 bytes. This is due to the byte by byte calculations performed in the inner loop of the algorithm. For our initial experiments we decided to use the BSD “best-case” size.

The sorting of the records was performed with the last name field as the primary key, the first name as the secondary, and the age field as the tertiary key. There were no duplicates in any of the data sets. Two different Predator versions were tested. The first is an array of 100,000 records. The second is a more complex data structure called a *segment*, and is explained in detail in Section 5.2.3.

Each run of the program produced a different randomization of the elements, and thus a different execution time. The initial comparison was performed on 48 byte records, with the size of the data set varying from 1,000 records to 110,000.

Each of the numbers shown in this graph resulted from 10 executions of the program on a given data set size. All of the standard deviations were well within one sigma.

The Predator and BSD versions both demonstrated the same asymptotic complexity. The ratio of their execution times varied by (at most) 2.5% when ranged over three orders of magnitude. This is very important. If the base case selection (or pivot selection) differed in the slightest, they would have different complexity classes, and there would be a discernible differential in the ratio of their executions as the number of records was increased. By modifying the Predator version to use the same base case and pivot selection as BSD, we are able to say that we are comparing the same program - not just two different implementations of quicksort.

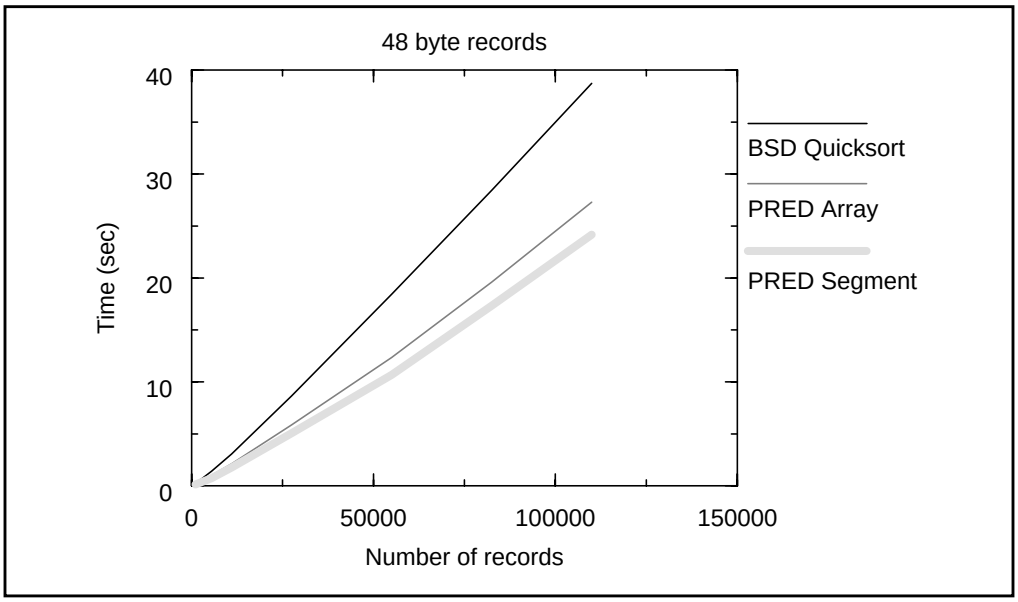

Figure 5.1: Quicksort of 48 byte records

The other lesson from this experiment was one of performance. Measured with a record size of 48 bytes, which should be BSD quicksort's best, Predator's array implementation of quicksort was approximately 30% faster than BSD quicksort. The ratio varied from a low of 67% to a high of 71%. Note that there is not a growing difference of ratio points - the ratio fluctuates about a mean of 69%. The segment implementation ran, by contrast, some 41% faster than BSD quicksort (on average). Its ratio varied from 57% to 62%.

The results of this test suggest that, even for production level, highly-optimized code, that the prototype Predator compiler generates competitive versions.

5.2.2 Varying the structure size

The BSD version of quicksort claims that it is optimized for 48 byte structures. This leads to the following question: How does changing the record size affect the speed difference between the Predator and BSD versions? Tests were run on both the BSD and Predator versions in which the number of records sorted was kept constant, while the number of bytes per record was varied. Two different tests were run. In the first, 1000 records were sorted. In the second, 99,000 records were sorted. This was done to ensure that, given two orders of magnitude difference, the same behavior would be observed as the size of the records was increased.

In each test, the size of the records was varied from a low of 4 bytes per record to a high of 200 bytes per record (over one order of magnitude). The results of those tests are shown below.

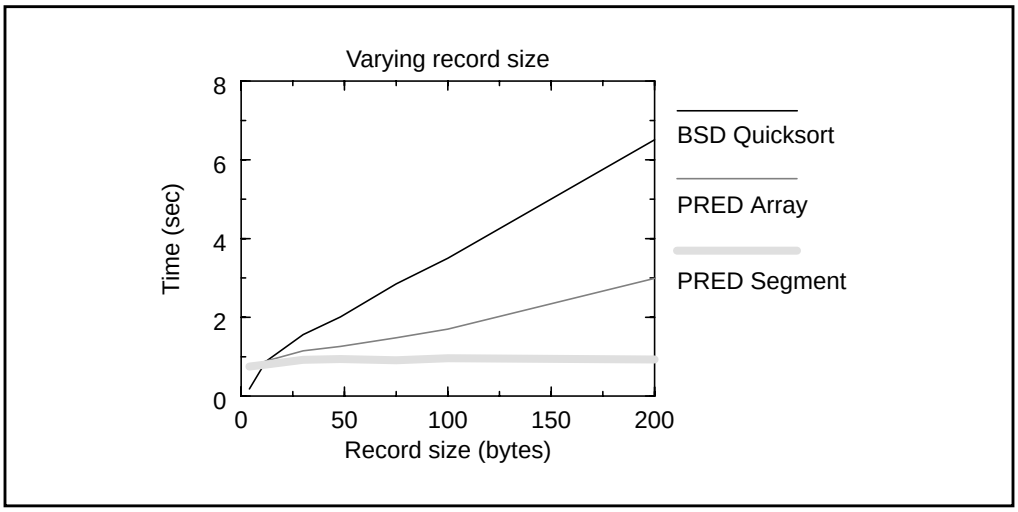

Figure 5.2: Quicksort of 1000 records

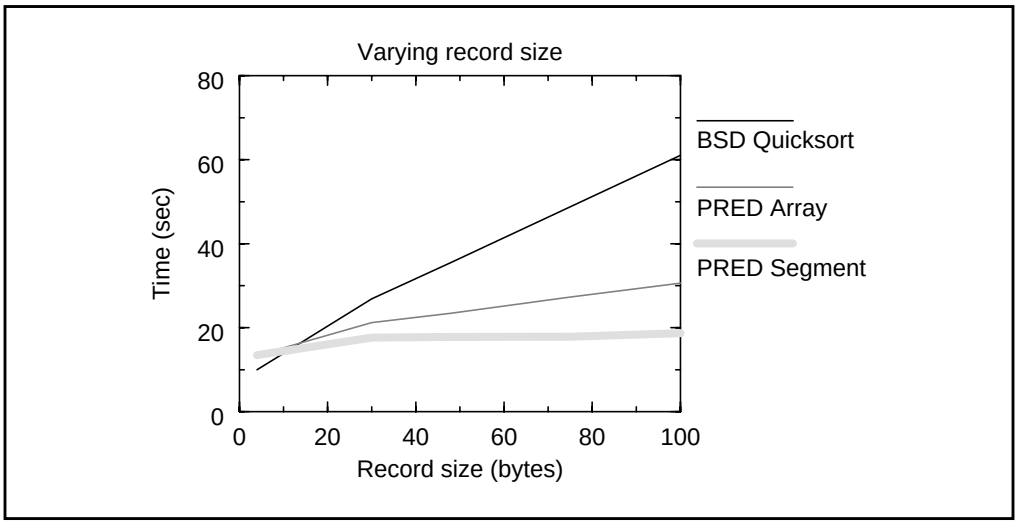

Figure 5.3: Quicksort of 99000 records

It is interesting to note from these tests is that as the number of bytes per record increases, Predator performs better and better relative to the BSD implementation. The reason for this is simple: the BSD implementation uses a byte-by-byte copy mechanism for record swapping. Predator uses a "whole record copy" scheme (i.e. **`rec1 = rec2`**). The Predator approach has a start-up cost, which causes it to be less efficient for small record sizes. The

per-record overhead, however, is fairly small. The BSD version, on the other hand, has a smaller start-up cost, but a much higher cost per byte copied. Thus, there is a crossover point at which the two approaches have comparable efficiencies. This occurs at a record size of 8 bytes. In the many cases in which records are larger than 8 bytes, Predator will provide better performance than BSD quicksort.

5.2.3 Segmented sorting

In all of the results shown above, there is a second Predator version shown: segmented sorting. The reason for investigating a segmented version of quicksort is simple. The BSD version of quicksort is capable only of sorting elements stored in a static array. And, as shown above, the time to sort an array increases linearly with the size of the record. This is due to the fact that there is simply more data to swap when one has larger records.

If one could somehow keep the size of the data swapped constant, no matter how large the actual data record became, it should result in superior performance for large record sizes. That is exactly what Predator segmented records do.

For the segmented case, we defined the data record as follows: segment each data record into two parts. The primary segment contains *none* of the fields of the original data record. This primary segment is stored in the static array. The secondary segment contains all of the data of the record, and is not stored in a data structure, but is simply *malloc'ed* on the heap. The Predator declaration for the segmented structure is:

```
SCHEMA cust_ds ON ELEMENT customer =
       segment[array[100000],
       ,
       malloc[]];
```

The second argument to the segment layer is the field at which to split the data records. In the example above, no such field is declared. Thus Predator places no data fields in the first segment, and all of the data fields in the second segment.

A sample populated array of data might look as follows:

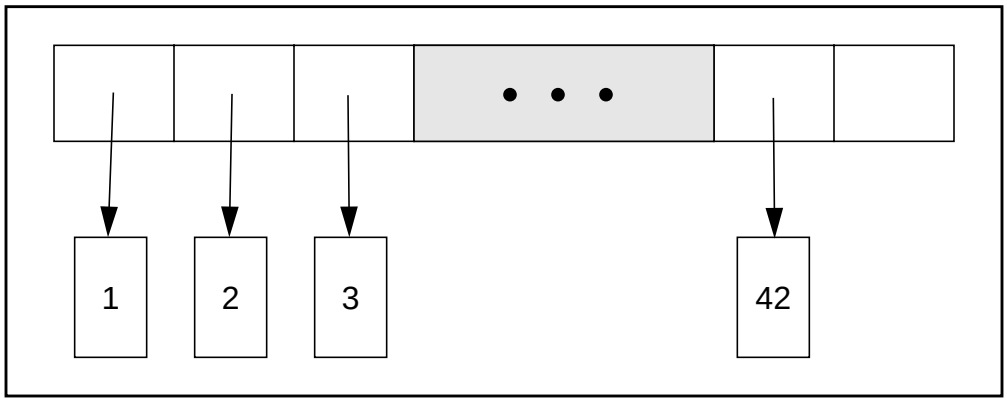

Figure 5.4: A segmented quicksort array

The pointer in the primary segment (pointing to the secondary segment) is automatically allocated and maintained in the Predator code. As far as the application programmer is concerned, they have, at the abstract level, a simple container of data.

The advantage to this data structure is that swapping is cheap (no data is ever swapped). Thus, performance is constant with respect to the size of the data record, significantly improved over the standard array-based quicksort. Further, since all of the segmentation code is automatically generated, only one line of code in the Predator program needed changing, which took less than 30 seconds. Again, this demonstrates the power of the Predator approach.

It is true that segmentation can be added to the BSD quicksort code. However, it is something which they *chose* not to add to their code. As part of this work we did, in fact, alter the actual BSD quicksort routine to work with segmentation. Due to the complexity of the BSD code, the effort took many hours, and even then we were not able to match the execution time of the Predator-generated version (Our modified BSD quicksort was slower than the Predator version). The ability to change data structures quickly is a real productivity benefit.

5.2.4 The *rwho* utility

There exist many applications which rely on the BSD quicksort routine. Some common UNIX utilities (such as **`ls`**, **`df`**, **`rwho`**...) consist of little more than data collection (into an array), sorting, and reporting. As a further exploration of quicksort we modified the **`rwho`** utility to work with Predator.

rwho is a utility which prints a list of users who are currently logged on to machines in a local network. It obtains this information from a data file, stores the data in an array, sorts the array, and then prints the output data in one of a number of formats, based on command-line switches.

The **rwho** program will fail if the number of user entries exceeds 100. This is because **rwho** stores the data in a static array (of size 100) so the data can be sorted by the BSD quicksort routine. This is an example of a feature existing due to the limitation of the data structure module, rather than due to an inherent limitation of the system.

It is possible to increase the size of the array, but that does not solve the problem, rather it delays it. We modified the **rwho** program to work with Predator, and then changed the underlying data structure to be a linked list. Since linked lists are unbounded (up to the available memory of the machine), the limitation is removed. The modified **rwho** program is, in fact, slightly faster than the original, and has the arbitrary restriction removed.

The total time required to investigate the **rwho** program, rewrite it to use Predator, substitute lists for arrays, and measure the performance was less than one hour. One could further augment **rwho** to operate even faster in certain cases. **rwho** provides the *-h* switch, which sorts and prints users by host names. For this option it might be desirable to create a version of **rwho** that uses a different container implementation (such as hash tables or binary trees). With Predator, the change would involve no more than changing a single declaration, and recompiling.

This experiment shows that the data structure independent nature of Predator allows programmers to avoid arbitrary restrictions on program features and functions normally imposed by tying the program to a specific data structure.

5.2.5 Quicksort conclusions

The investigations performed with the BSD version of quicksort were illuminating. The following observations can be made:

1. The Predator version more closely resembles the actual high-level algorithm. It is easier to read and debug[3]. In addition, new features for improving the algorithm (such as

3. The source code for both versions can be found in Appendix C.

adding segmentation) are easier to add to the Predator version. The BSD version is so highly optimized, that modifying it is difficult.

2. For the large majority of useful record sizes, the Predator version is faster than the highly-tuned BSD version. The Predator version is also easily modified (with the segment layer) to be even faster, which improves the complexity class of the solution.

3. The Predator version can be made to work on virtually *any unordered* data structure in a matter of minutes. This can be useful in adding new features, and can also allow for programmer experimentation to discover the best choice of data structure for a given data set. This contrasts with BSD quicksort, which is tightly coupled to the array data structure.

We believe that the results we obtained with BSD quicksort are not at all unusual. We selected quicksort because we felt that if any function was commonly used and highly optimized, it would be sorting. We selected BSD quicksort, specifically, because of the length of time it has been in the public domain. Many other basic algorithms exist (such as Ziv-Lempel encoding) that might have been equally good candidates. It is our belief that the benefits shown here are not uncommon and will be true of many other basic code functions currently used.

5.3 Comparing against data structure libraries

Many libraries of data structure modules are currently available for programmers to use when writing their applications. It is logical to ask, therefore, how Predator-generated code compares against that available in those libraries. This section reports on a benchmarking effort, which is reported in greater detail in [Bat93a].

A spell checker benchmark was written. When invoked, the program reads in a dictionary file of 25,000 words, in randomized order, which it places in a data structure. Next, a document is read in from standard input. Duplicates are discarded, and each unique word is placed in a second data structure (of the same type as the first). Finally, the second data structure is scanned, and each word is looked up in the dictionary. Words which are not found are then printed out to standard output.

Standard input and output routines were written, and used in each of the programs. Three different library systems (Booch components for C++ [Boo90], libg++ [Lea88], and Pred-

ator) were benchmarked. Each of the systems was benchmarked against unsorted and sorted arrays, unordered lists, and ordered binary trees. The Predator source of the array version can be found in Appendix D.

Table 5.5: Size of benchmark programs

Component Library	Unordered list	Unordered array	Sorted array	Binary tree
Booch C++ Components 2.0	320 words	360 words	398 words	481 words
libg++ 2.4	336 words	386 words	474 words	336 words
Predator	281 words	281 words	287 words	285 words

Each of the programs was executed against a standard document (the Declaration of Independence), and timed. Each program was executed 5 times, and the mean values were rounded to the nearest second. This level of accuracy is sufficient to illustrate the differences among the programs[4].

It can be seen from Table 5.5 that the Predator versions of the benchmark are signifi-

Table 5.6: Execution speed of benchmark programs

Component Library	Unordered list	Unordered array	Sorted array	Binary tree
Booch C++ Components 2.0 (compiled with Sun CC 3.0.1 -O4)	70.9 sec	54.6 sec	11.1 sec	15.4 sec
libg++ 2.4 (compiled with G++ 2.4.5 -O2)	41.9 sec	34.3 sec	5.4 sec	4.1 sec
Predator (compiled with GCC 2.4.5 -O2)	40.2 sec	33.3 sec	6.3 sec	3.0 sec

cantly smaller than those written for the other libraries. When it is considered that 211 of the words in all of the programs are the common I/O routines, one can see that the data structure code in the Predator versions is rather concise.

More importantly, it took a mere few minutes to change the type equations to modify the Predator array benchmark to create the other Predator versions, whereas recoding the bench-

4. Two of the three libraries were compiled with the gnu C/C++ compiler. The Booch library requires the Sun C++ compiler (for library compatibility).

mark for a new data structure (for both Booch and libg++) entailed considerable effort (due to the fact that their interfaces differ for each data structure). Not only did Predator save time for the benchmark programmer, but (in the future) additional data structures can be easily tested with Predator, simply by changing the type equations.

In the future, we hope to expand both the number of data structures measured and the number of data structure libraries[5]. Also, a project is underway to construct an automatic data structure benchmark generator. This generator will be capable of generating code (via a script language) to test many common data structure functions (such as insert, delete, update, scan, search, retrieve etc.) for many different software component libraries.

5.4 Emulating a production system compiler

Our acid-test validation was to verify that our approach could scale well to a system which contains a large number of data structures and code modules. We felt that a good candidate system should:

1. be sufficiently large.
2. include large amounts of data structure code.
3. be complete, so that performance data would exist for comparison purposes.
4. be state-of-the-art in its field.
5. be a well-written system.

We selected the *LEAPS* production system compiler project. LEAPS [Mir90, Mir91] (Lazy Evaluation Algorithm for ExPert Systems) is a production systems compiler which converts OPS5 rule sets into a C-language program, which can then be compiled, linked, and run. In simpler terms, LEAPS takes a general description of a production system (rules) and produces an imperative version of that production system. The executable can then be run against initial data sets. LEAPS offers many advantages for emulation:

1. It is a mature project with many performance results published.
2. It is a state-of-the-art system with results that compare favorably against other systems in its field [Mir90].

---

5. In the near future we hope to add additional libraries [Gor87, Fon90] to our benchmark.

3. It contains a large number of data structure instances (containers) in its run-time library.

4. It uses many different data structures (schemas) including lists, stacks, predicate indexes, and links.

5. The code that implements LEAPS is very complicated. Debugging of this system (mainly due to errant pointers) has been reported to be very difficult. Further, certain new features have not yet been added due to the complexity of the existing code and the difficulty of integrating new features.

6. LEAPS is a University of Texas project. Technical assistance was available.

Given all these advantages, we decided to take several LEAPS test cases (rule sets), emulate them with Predator equivalents, and study the results.

5.4.1 OPS5

OPS5 [For81] is a language for describing production (expert) systems. It is a mature system, which is used widely in the AI community. OPS5 is a mature language, and LEAPS is but one of many approaches [McD78a, McD78b] designed to evaluate OPS5 programs. OPS5 syntax is based on LISP, and OPS5 is a dynamically-typed language. This has serious ramifications which will be explained later. For now, we will only discuss statically-typed issues of OPS5.

There are a few basic concepts of production systems which must be defined. The *database* is the collection of data for the production system, and consists of a set of *data tuples*, each of which is a n-ary relation of data items.

Using Predator terms, one can envision a database as being a set of containers. Each data tuple is an element which is a member of a particular container, and which is defined by a specific schema. For example, a database for a sample production system might consist of data tuples from two containers: a container of customers and a container of departments. The following could represent the state of the sample database at a given time:

```
{{name = Joe Smith, id = 3, dept_# = 42},
 {name = Sam Spade, id = 6, dept_# = 23},
 {dept_num = 42, name = A department, num_employees = 3},
 {name = Marion Librarian, id=7, dept_# = 42}
}
```

The database above contains three employee tuples, and one department tuple. The order in which they are shown is of no consequence.

A basic construct in OPS5 is the *rule*. A rule is constructed of two parts: the *condition* and the *action*. A condition is a set of boolean predicates joined together conjunctively. Each individual *condition element* (or CE) is a predicate performed on some part of the database for the system. An action is a set of executable clauses which can be performed on the database.

A sample rule is shown below:

```
(p first_rule
  (employee ^id 4)
-->
  (write "We found employee number 4!!")
)
```

This is a very simple rule (or production), whose name is *first_rule*. This rule contains one condition element. That CE tests to see if there is any employee present in the container *employee* whose *id* field is equal to 4. If there is one that satisfies this condition, the action set for the rule is fired. In this case, a text string is written to the standard output.

A rule is *satisfied* if one or more elements exist in the database which will cause the condition of the rule to evaluate to true. More than one rule can be satisfiable at a given time. It is up to the algorithm implementing OPS5 to select which of the satisfiable rules will be executed and in what order.

A more interesting rule is:

```
(p second_rule
  (employee ^dept_num <x> ^name <y>)
  (department ^dept_num <x> ^num_employees <z>)
-->
  (modify 2 ^num_employees (compute <z> + 1))
  (write <y> " is in the join in department " <x>)]
  (remove 1)
)
```

This rule creates a join between the employee and department containers. For each employee and department pair where the *dept_num* fields are the same, the *num_employees* field of the department record is incremented, an output message is printed, and the employee record is deleted. The numbers in the remove and modify actions refer to the condition element number in the condition section of the rule. For example, **(remove 1)** means that the record referred

to in the *first* condition element (the employee element with **dept_num == <x>**) will be deleted from its container.

Containers are defined with the *literalize* statement. This statement is optional, but is used in all of the OPS5 rule sets given to us for evaluation. A sample literalize statement might look like:

```
(literalize employee
            name
            id
            dept_#
)
```

Literalize is a very simple construct. It does nothing more than declare the name of a container, and the field names for an element of the container. Note that no types are shown. It was mentioned earlier that OPS5 is dynamically typed. There are three basic types in OPS5: integer, float, and string (symbol). Any field of an element may, at any time, contain an entity of any of the types. In reality, a given field usually only contains values of a specific type. All of the OPS5 rule sets we were given to emulate were statically typed. Our automated RL compiler (currently under construction) extends the Predator version of OPS to handle dynamic typing.

The final major OPS5 concept that must be understood is the *production system* (PS). A production system is a set of OPS5 rules. To execute a PS, one populates the database with initial data values, and executes a *production cycle*. A production cycle consists of locating a satisfiable rule and firing the actions for that rule. The actions may change the state of the database, which may affect the rule selection for the next cycle. When the system arrives at a state where no rules are satisfiable, the database is said to be at a *fix point* (steady state), and the program terminates.

The model we used for OPS5 programs is as follows:

```
load in initial data into database;
while (there is a satisfiable rule) /* match/conflict resolution*/
{
  execute the action for the selected rule;
}
```

One iteration of steps through the while loop above constitutes one production cycle. Within the production cycle, match/conflict resolution is the crucial step. Exhaustively searching the database for a satisfiable rule (on each production cycle) is relatively easy to program,

but is grossly inefficient in time. LEAPS has improved on classical OPS5 systems by devising a good method for representing the state of the database, and using that representation to locate efficiently a good candidate satisfiable rule.

5.4.2 LEAPS

LEAPS is a compiler which converts an OPS5 production system description into a C program. That program is then compiled into an executable image. LEAPS adds a driver-like front-end to each production system to allow for initial data to be loaded, commands to be run and primitive debugging to be performed. A sample LEAPS authoring cycle is shown in Figure 5.5.

In the LEAPS system, the database is also called the *working memory. Working memory elements* (WMEs) are data elements (tuples). WMEs contain a class name (a Predator schema name) and a set of attribute-value pairs. This structure for WMEs owes much to the LISP origins of OPS5.

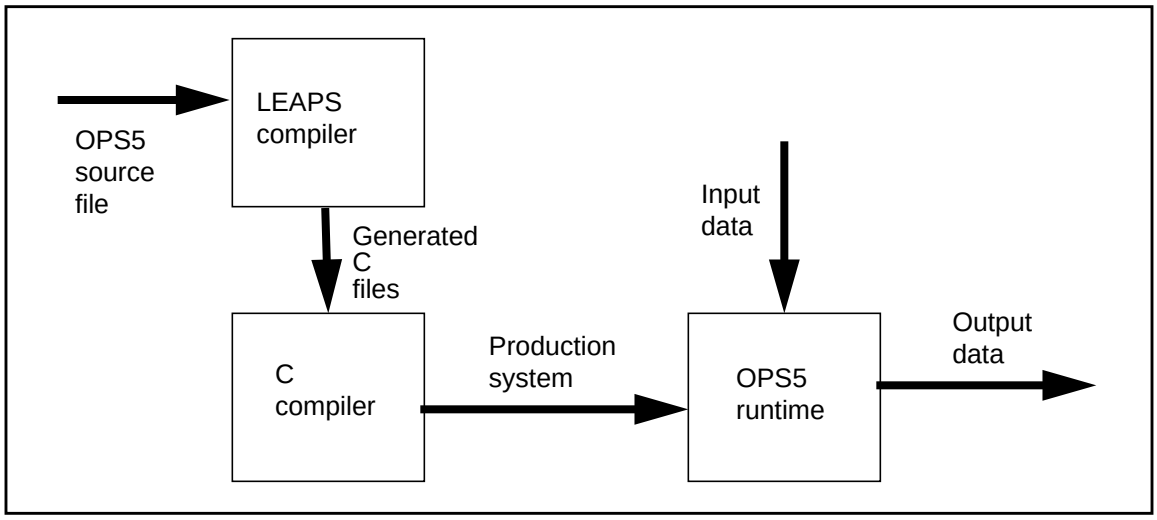

Figure 5.5: A LEAPS authoring cycle

Each WME also contains a timestamp. A timestamp is an integer that denotes the cycle number in which the object was last written (created or updated).

The *wait list* is an ordered linked list, which LEAPS uses to reference WMEs quickly in timestamp order. Each time a WME has a write access (for creation, update, or deletion), it is either added to, removed from, or moved within the wait list.

Alpha memories (α-memories) are another important concept. An α-memory is a chained indexing method used to reference WMEs which satisfy a specific predicate. It is possible, under LEAPS, for many different α-memories to be defined on a single container.

The *dominant objec*t for a given cycle is that object (element) in the database which has the most recent timestamp (this definition will be expanded somewhat with the inclusion of joins below).

It was mentioned above that matching and conflict resolution is the most difficult phase of a production cycle. This is due to the fact that in most cycles many different k-tuples of objects exist (O_1, O_2, ... O_k) which could satisfy a rule in that cycle. The gain to be made from an efficient algorithm is to find a satisfiable rule in the least amount of time. LEAPS is based on lazy evaluation. The basic match/conflict resolution algorithm is as follows[6]:

```
while (there are still objects on wait list)
{
  remove dominant object from the wait list;
  do
  {
    locate next most specific rule whose predicate
      references dominant object;
  } while (rule exists && rule cannot be satisfied with
           elements in working memory);
  if (rule was found)
     exit match/conflict resolution; /* exit while loop */
}
```

A rule **R1** is more specific that a rule **R2** if **R1** references more containers than **R2**. If two rules are equally specific, then LEAPS first tests the rule which occurs earlier in the source text, and then the rule that occurs later in the source text.

It so happens that all of the basic concepts described above can be mapped to relational equivalents. Table 5.7 shows the correspondences.

---

6. This is not the complete LEAPS match/conflict resolution algorithm. In succeeding sections, we will augment the basic form of the algorithm shown here.

Table 5.7: Relational equivalents of OPS5/LEAPS concepts

OPS/LEAPS concept	**Relational Equivalent**
Class	Container
WME	Element
Attribute	Field
Database	Collection of containers
Condition Element	Conditional test
α-memory	Predicate index
Production System	Program
Rule	Conditional test/action
Join	Link/join

5.4.3 The RL driver

A sample driver for RL (Re-engineered LEAPS) was constructed. In comparing the efficiency of the RL code versus the LEAPS code, it was important for the comparison to be fair. It is not sufficient that the RL version read the same input and produce the same output as the LEAPS version. To ensure that the LEAPS algorithm was followed faithfully, our RL driver had to perform exactly the same work as the LEAPS program. The following five conditions are necessary:

1. The RL driver had to read the same input data files as the LEAPS version.
2. The RL driver had to produce *exactly* the same output as the LEAPS version[7]. This required (among other things) that the RL driver write the LEAPS copyright statement to standard output!
3. The RL driver had to fire the same rules in the same order as the LEAPS version.
4. The RL driver had to investigate the same dominant objects in the same order as the LEAPS version.
5. The RL driver had to both retrieve k-tuples and test the rules for satisfiablity in the same order as LEAPS.

Before any of the speed/performance tests were run to compare LEAPS and RL, a set of compatibility tests were run. For all of the tested production systems (and for all input data

7. Comparisons were made with the UNIX *diff* program to ensure byte-by-byte matches.

sets for those PS), both LEAPS and RL were placed in diagnostic mode. In this mode, a dump was made of each dominant object, k-tuple, rule tested, and rule fired in the system. The dumps for RL and LEAPS were then compared. Thus, the RL driver, for every production system we tested, does *exactly* the same work, in the same order, as the LEAPS production system it is emulating.

After we verified that our RL programs performed exactly the same work as their LEAPS' counterparts, we removed all **writeln** output statements from both programs, and recompiled. We did this to base our timing comparisons, as much as possible, on the actual RL and LEAPS algorithms and code, not on the system-supplied output routines.

5.4.4 Emulating simple production systems

The match/conflict resolution algorithm given above is sufficient to execute simple production systems. Simple production systems are those in which rules have a maximum arity of 1. While it is difficult to produce a simple production system which does much that is interesting, it was useful to write (and test) several simple production systems to ensure that a proper RL driver was constructed.

Three simple rules sets were evaluated and run; all displayed similar behavior. One of those, *nj1.ops* (no joins), will be described in detail here.

nj1.ops is a single-rule production system, which does nothing more than count from a starting value (typically 0), up to some upper bound. In essence, it is nothing more than a production system version of a for-loop. By varying the upper bound of the loop we were able to compare the efficiency of the Predator and LEAPS versions of the system. The following is the no-join1.ops rule set:

```
(literalize foo
 bar)

(p no-join-test
 (foo ^bar {<x> < 500})
-->
 (write "bar has value" <x>)
 (modify 1 ^bar (compute <x> + 1))
)
```

The single rule (no-join-test), has a single CE, which does nothing more than test to see if the **bar** field of a **foo** element has a value less than 500. If it does, an output statement is

written, and the **bar** field of that element is then incremented by one. The initial data for this PS is a single **foo** element with a **bar** field of 0. Successive production cycles cause the value to be incremented until the element has a **bar** value of 500. At that point, the dominant object satisfies no rules, there are no further dominant objects, and the system reaches a fix-point and terminates.

The following graph shows the performance of both the LEAPS and RL versions of no-join1, for a range of maximum values.

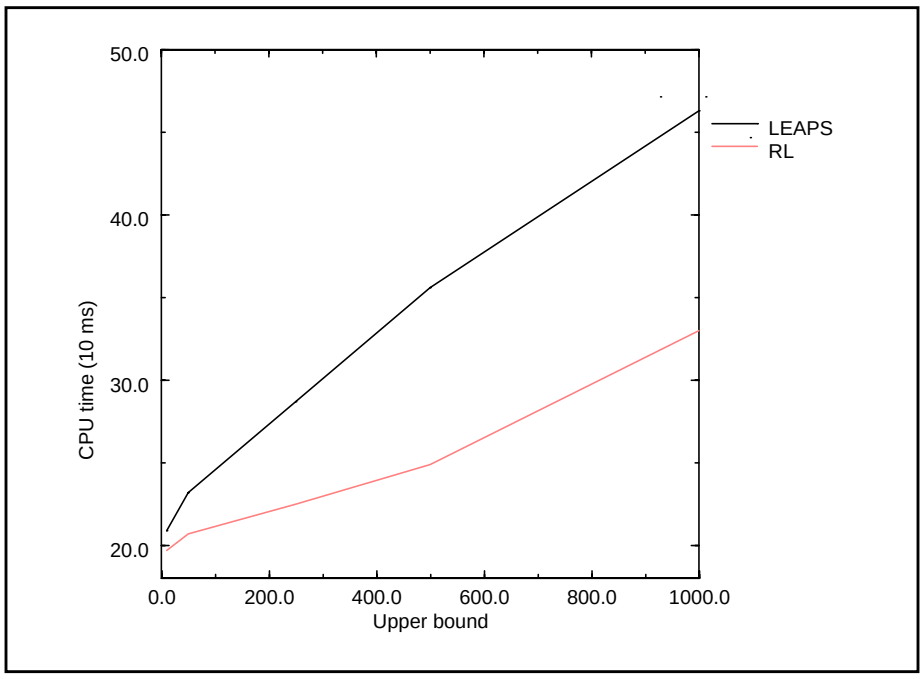

Figure 5.6: Execution speed of no_join1.ops

The difference in execution speed was somewhat surprising, as we found that they were both linear, but with different slopes. There are many contributing factors, but the largest is that LEAPS is based on LISP, and causes each data item to be stored in a CONS cell. Traversing these structures (as opposed to simple C malloc'ed structures for RL) is not particularly efficient.

Two other simple rules sets were translated to RL and evaluated. The OPS5 code for those systems (as well as all other PS tested) can be found in Appendix B. The results of testing these other simple PS show similar results to that of no-join1.

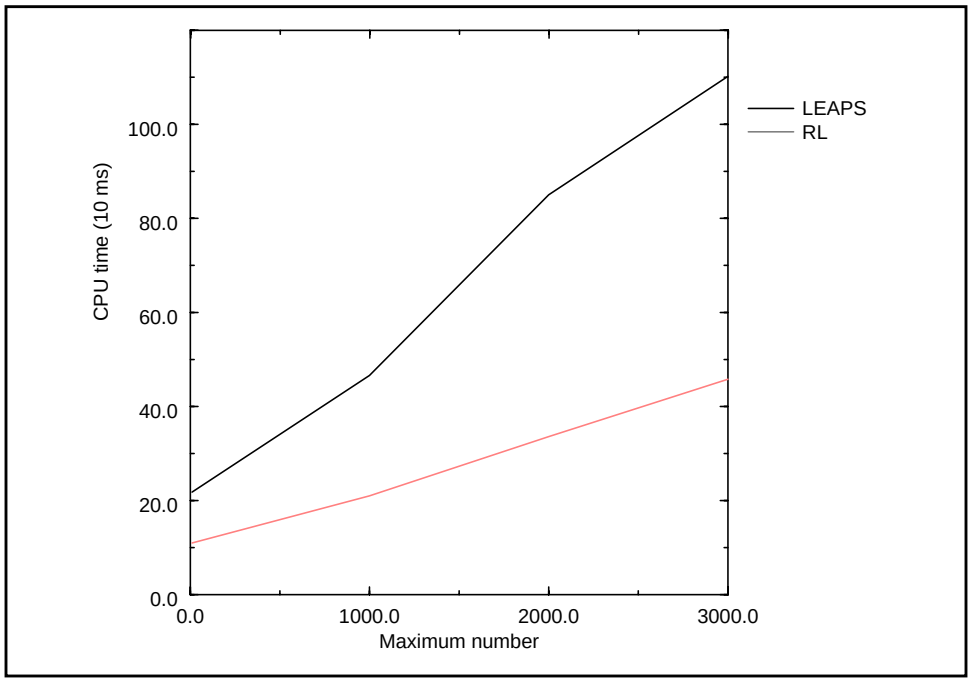

Figure 5.7: Execution speed of no_join2.ops

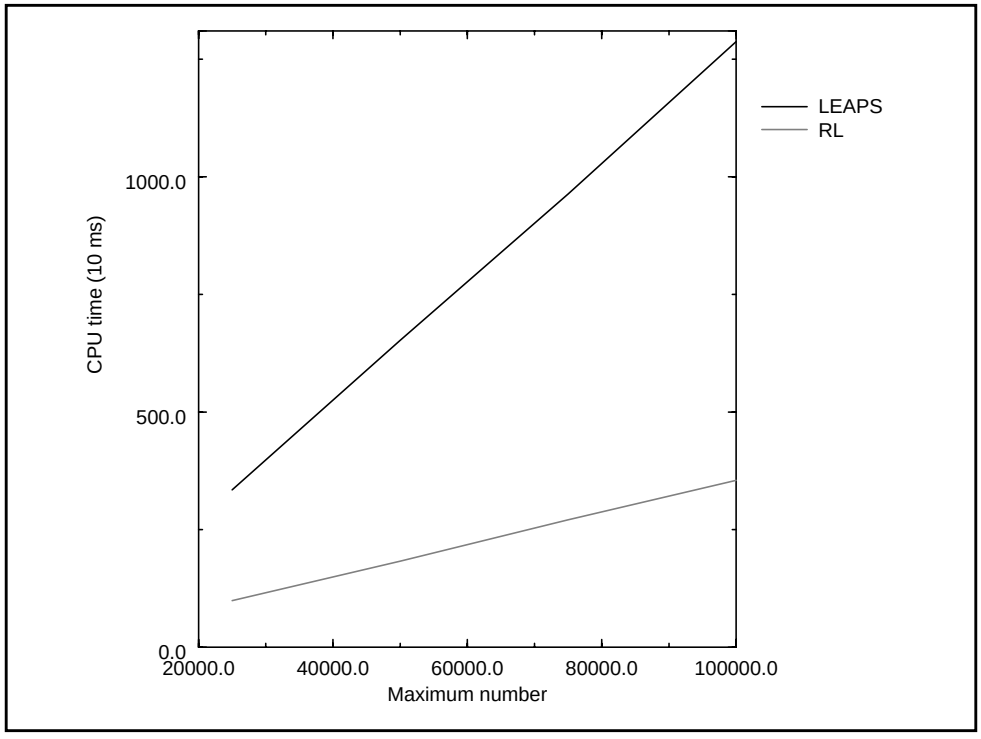

Figure 5.8: Execution speed of basic_cycle.ops

In summary, for simple wait-list production systems, Predator generates more efficient code. Useful production systems, however, require additional features.

5.4.5 Production systems with joins

Most useful production systems require joins. Any time a rule contains two CEs which reference more than one element from one or more containers, it implies a join.

Joins are an area where LEAPS has made significant improvements over other OPS5-based systems. LEAPS defines a *wait-stack* in addition to the wait-list. The wait-stack is a stack of partially completed joins, ordered by timestamp[8]. Every time that LEAPS finds a rule to satisfy that utilizes a join, it fires the rule on the first tuple of that join, and then places the state of that join on the wait-stack. The timestamp of a wait-stack entry is the timestamp of the dominant object of that join. At a later time, the state of the join will be restored, and the execution cycle will determine if the join produces any further tuples which satisfy the rule. If one does, the join is again placed on the wait stack for later execution. This lazy algorithm differs from a more classical approach, in which a join, once started, must exhaust all of its possible tuples before any other dominant objects may be investigated. The advantage of this approach is that the chain of execution of one tuple of a join may cause state changes which remove branches from the join's search tree. In other words, it delays action on tuples which may not need to be ever considered.

The basic execution cycle of RL must be modified for the addition of joins. The match/conflict resolution phase is augmented.

```
while (there are still objects on wait list and/or wait stack)
{
  remove dominant object from the wait list or wait stack;
  if   (dominant object is from wait list)
       do
       {
         locate next most specific rule whose predicate
           references dominant object;
       } while (rule exists && rule cannot be satisfied with
                elements in working memory);
  else
```

8. In actuality, the wait-stack is used in simpler production system for partial results. It is described in this section only to avoid making the original explanation too complicated.

```
    {
         restore state of partial join;
         locate next tuple that satisfies the selected rule;
    }
    if (rule was found)
    {
       if (tuple found is part of a partial join)
          place state of join on wait stack;
       exit match/conflict resolution; /* exit while loop */
    }
}
```

The basic execution cycle of LEAPS is now augmented so that, on a given cycle of the program, LEAPS looks at both the wait-stack and the wait-list. The object that is selected as dominant is the object which has the greater timestamp of the two. If it is the wait-list object, LEAPS continues as before. If the dominant object resides on the wait-stack, LEAPS restores the state of the join, and tries to find another tuple which satisfies the join. If one exists, the join is returned to the wait-stack and the satisfying rule is fired. If the join produces no further satisfiable rules, it is removed from the wait-list and another basic cycle begins.

LEAPS introduces another optimization for rule conflict resolution. When considering a given rule **R** with condition element set **E**, LEAPS breaks down set E into two sets, $\mathbf{E_1}$ and $\mathbf{E_2}$. $\mathbf{E_1}$ consists of all condition elements which reference the dominant object. $\mathbf{E_2}$ consists of all the other CEs (that reference other elements). $\mathbf{E_1}$ is called the *local* predicate, and $\mathbf{E_2}$ is called the *residual*. Speed can be increased in the conflict resolution phase in the following way: Suppose that a dominant object **D** is selected, and a rule **R** is the most specific rule to check. LEAPS first checks to see if **D** satisfies the local predicate $\mathbf{E_1}$. If it does not, there is no reason to check the residual. If it does, then the residual is checked. Note, also, that the residual is now a join over *n-1* containers, whereas **E** was a join over *n* containers (i.e. the arity of the join is reduced by 1).

Note that while this can save considerable time, it does generate additional code. LEAPS must generate a local and residual predicate for each container type that participates in the condition elements for each rule. RL fully supports this optimization. Each rule satisfaction procedure is broken down into local and residual sections. Further, RL has one satisfaction function for each container type for each rule in the PS.

Join algorithms are important in LEAPS. The standard LEAPS algorithm uses simple nested loops for join implementations. In Predator, links are nothing more than another realm (which is a subtype of **DS**). Predator currently implements multiple link implementations,

including nested loop. It was possible, therefore, to run multiple RL executions for each join test case, by simply changing the **LINK** statement in the RL driver, and recompiling.

The test results from the *triples.ops* test case follows. triples.ops is a PS which locates all triples **<x, y, z>** of integers less than **n**, where **x < y < z** (i.e., the sequence is strictly increasing). Two different versions of RL were written for triples.ops. One is a nested loop algorithm, the same as LEAPS uses. The other is a pointer-based link algorithm. For more information about pointer-based links, see Section 4.2.2.

A major advantage of RL over LEAPS can be seen from this test case. The link implementation for RL can be changed by simply changing the type equation and recompiling. Pointer-based links are more efficient for triples.ops than are nested loop links. This points out the flexibility of NPTs - the LEAPS programmers have wanted to add pointer-based links to their system, but because the coding of it was too daunting, it has not yet been added.

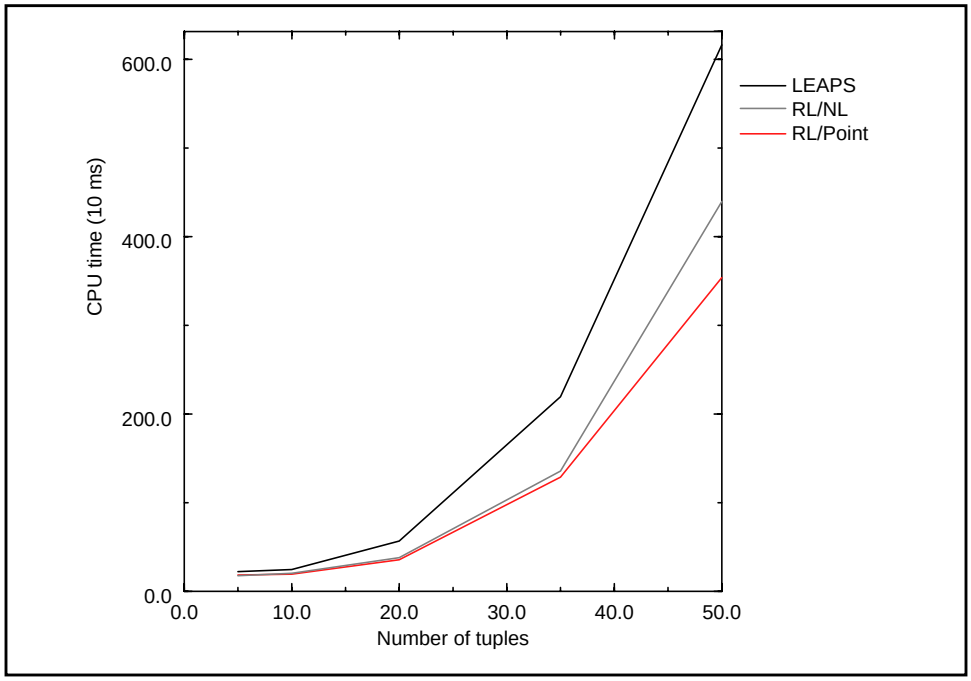

Figure 5.9: Execution speed of triples.ops

As can be seen, both of the RL versions of triples.ops outperformed the LEAPS version. The nested loop version of RL retains all of the performance advantages mentioned in the previous section.

The pointer-based method of links outperforms nested loop (for this test) because the cross product of the containers is traversed a large number of times. While the pointer-based method of links does have additional maintenance overhead when compared to the nested loop method, it generates the interlink connections when the elements are added to the containers. This overhead is dwarfed by the time saved due to the reduced number of traversals of the whole join. Note that in certain production systems, nested loop links will be more efficient than pointers. The advantage to the NPT approach is that the programmer may make that choice for each individual production system.

Two other join production systems were also evaluated. Again, they demonstrated similar results to triples.ops.

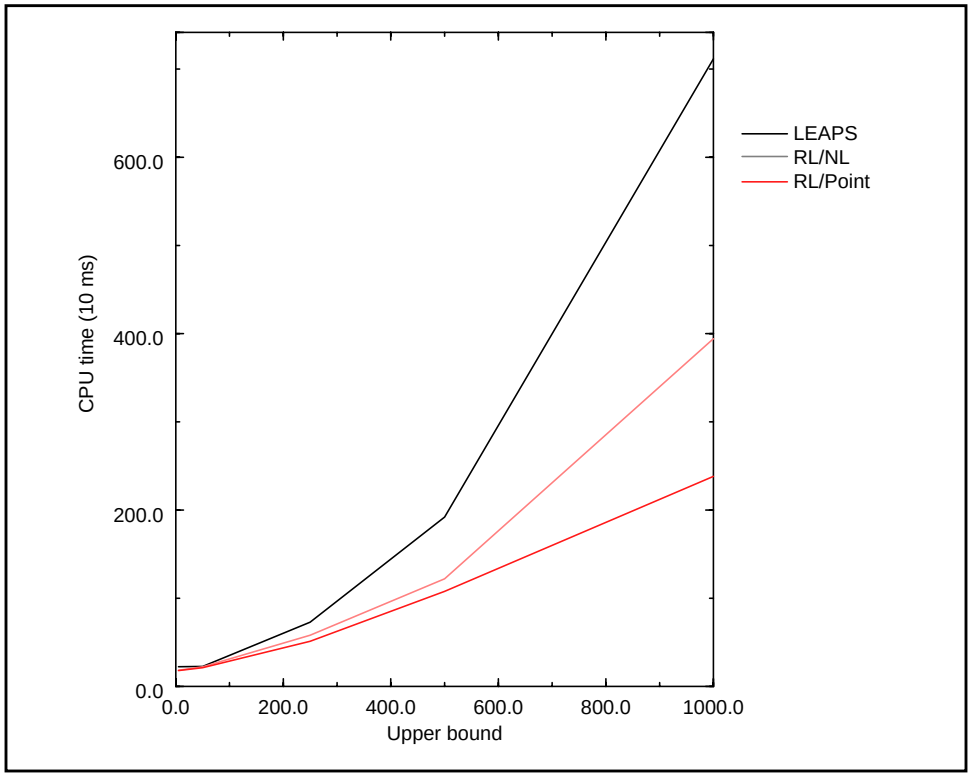

Figure 5.10: Execution speed of big_join.ops

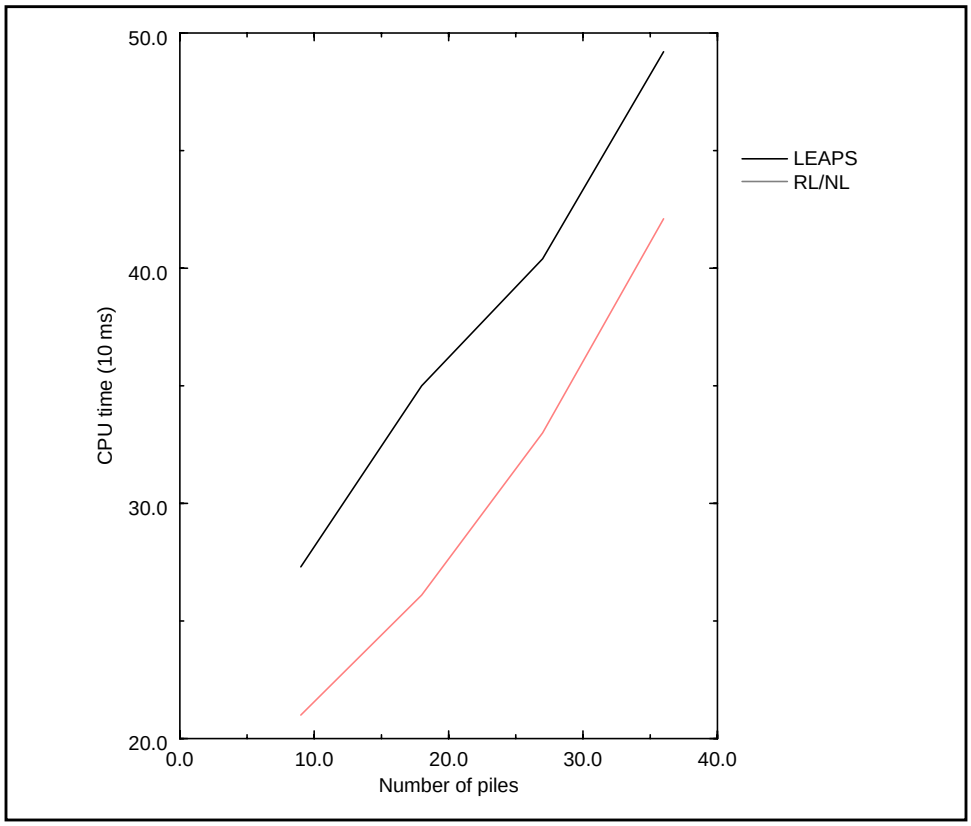

Figure 5.11: Execution speed of puz.ops

5.4.6 Adding negation to condition elements

Negation is another major feature of OPS5 that is implemented in LEAPS. Negation is the process of placing a – (does not exist) symbol in front of a condition element of a rule. For example:

```
(p neg_test
 (test_cont ^value <x>)
 (other_cont ^other_val = <x>)
 -(test_cont ^value > <x>)
-->
 (write "action here!")
)
```

The rule above will be satisfied for the element in the *test_cont* container which has the largest *value* field, and has a matching record in the *other_cont* container. The reason is that all elements in test_cont satisfy the first CE, but only one element can satisfy the condition that "there does not exist another *test_cont* element whose *value* field is larger than **<x>**.

While negation may be a difficult concept to grasp, it is relatively simple to implement in RL. Each time a join condition is evaluated, each negated CE is placed at the end (innermost location) of the residual for that join. For example, in the above rule, RL might generate the following condition test:

```
CURSOR c0 ON test_cont;
CURSOR c1 ON test_cont;
CURSOR c2 ON other_cont;

int check_neg_test(CURSOR c0)
{
  FOREACH(c2)
  {
    if (c2.other_val == c0.value)
    {
       FOREACH(c1)
       {
         if (c1.value > c0.value)
            goto next_c2;
       }
       return(TRUE);
    }
next_c2: ;
  }
  return(FALSE);
}
```

The code above is not actually what would be found in RL, but is a simplified version (RL would use links and composite cursors; too complicated to show here). The above example assumes that the three cursors shown are defined over the containers, and that the function is called with the dominant object already set in the *c0* cursor. It then checks all non-negated CEs (the one join clause). When it finds a join relation that satisfies the two clauses, it checks the negated clause. If an element is found that causes clause 3 to fail, another *c2* must be found. If no *c1* is found, the entire condition set is satisfied, and the function returns TRUE. If all c2 values fail, the rule is not satisfiable for that dominant object, and the function returns FALSE.

Because LEAPS utilizes lazy evaluation, negation causes an unusual side-effect. When a rule's action requires an element to be deleted, it may be impossible to do so. The element in question may still be involved in a suspended partial join that resides on the wait-stack, and cannot be purged (permanently removed) from memory until it no longer participates in any joins. If LEAPS were to purge it prematurely, incorrect output could result.

It so happens that, while this is a difficult feature to implement in LEAPS, it is not too difficult to add to RL. Consider the following type equation from Predator:

```
SCHEMA test ON ELEMENT elem =
       predind[delflag[predind[malloc[],
                               <condition>]],
               <condition>];
```

By placing two different predicate index layers *with the same condition* on either side of the delflag it is possible to allow RL to "delete" the element from the outer predicate index, while leaving it on the inner (because the delflag layer does not pass **DELETE** operations to its lower layers). This action causes any future new (wait-list based) searches to not see the element (since it has been deleted from the outer predicate index), but joins which are pulled from the wait stack can still locate the element based on the inner predicate index. When the element can no longer participate in any joins (when the item is marked for deletion, and the dominant object in the system has a smaller timestamp than the to-be-purged object), the element is purged, which removes it from both the inner index and memory. In LEAPS, this is a fairly complicated operation, and is still not fully implemented. Due to NPTs high-level nature, the algorithmic behavior was fairly simple to capture and code. This is another example of Predator simplifying the programming task. It is very simple to create the two predicate index layers and to allow Predator's query optimizer to select the proper layer to use for both types of scans.

Three different negated CE test cases were executed and compared. The results for the simplest of these, big_num.ops, is shown below. big_num.ops is little more than the example covered above.

Due to a problem in the prototype RL driver (self-referential links not implemented automatically for pointer-based links), only the results for nested-loop links are shown for these test cases.

As with other test cases shown, the RL driver code is more efficient than LEAPS, and shows a consistent linear improvement.

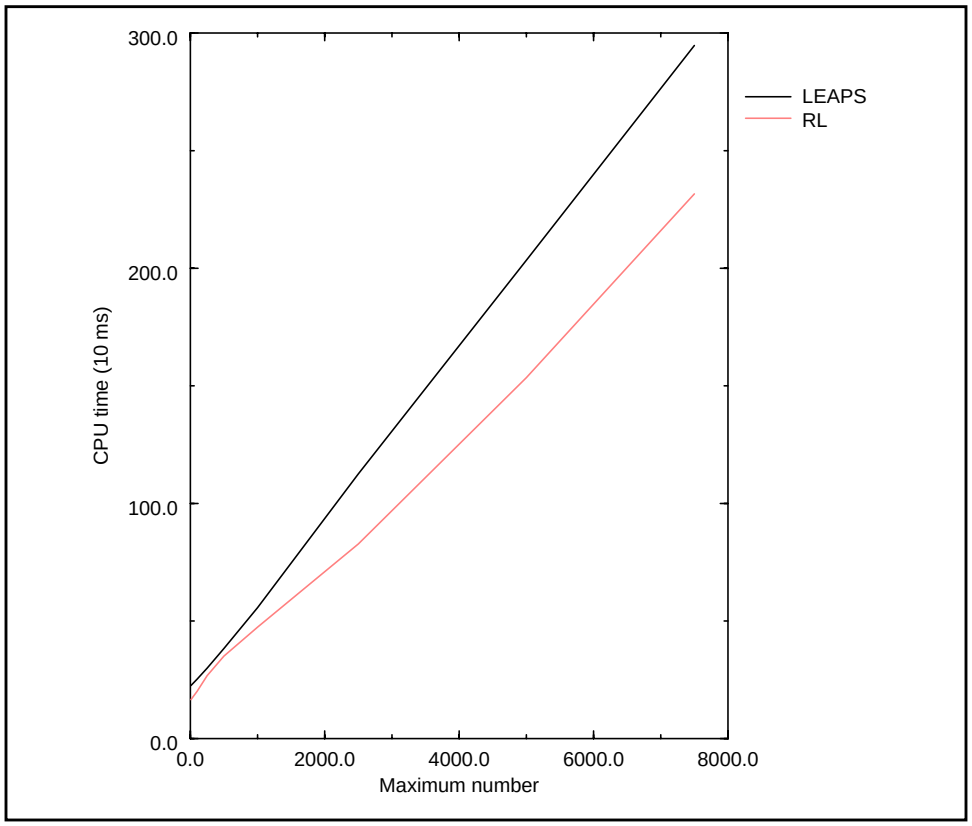

Figure 5.12: Execution speed of big_num.ops

The next test case, jig25.ops, demonstrates another strength of the Predator approach over LEAPS. jig25 solves a two-dimensional puzzle. What makes it interesting is, unlike the other OPS5 test cases, the input data is not stored in an input file, but in an initial "start-up" rule. In other words, the input data file contains one WME, which causes the start-up rule to fire. As we increased the size of the input data set (to measure performance of LEAPS vs. RL), we discovered that the time required to compile the LEAPS-generated C code was growing exponentially (whereas the RL code compile time grew linearly). At a size of 700 input items, the LEAPS program required 1 hour and 23 minutes to compile. At 800 items, the compiler core dumped after 2 hours and 30 minutes. The compile times are shown in the graph below.

We could only get the LEAPS implementation to compile for data sets larger than 700 items by moving the initial data from the program to a data file. Speaking to the LEAPS programmer, we found out that this is a major limitation of their approach. They add considerable code for each *make* statement that is located in the rules themselves. They have found no compiler that can handle large data sets in the rules[9]. RL, by contrast, had no difficulty scal-

ing to large data sets in the rules themselves. Each *make* (OPS 5 data assignment) statement in the OPS5 rule was replaced by a single function call to a routine that inserted the data item into working memory. This is a major limitation in the LEAPS compiler's design, that the Predator compiler handles with ease.

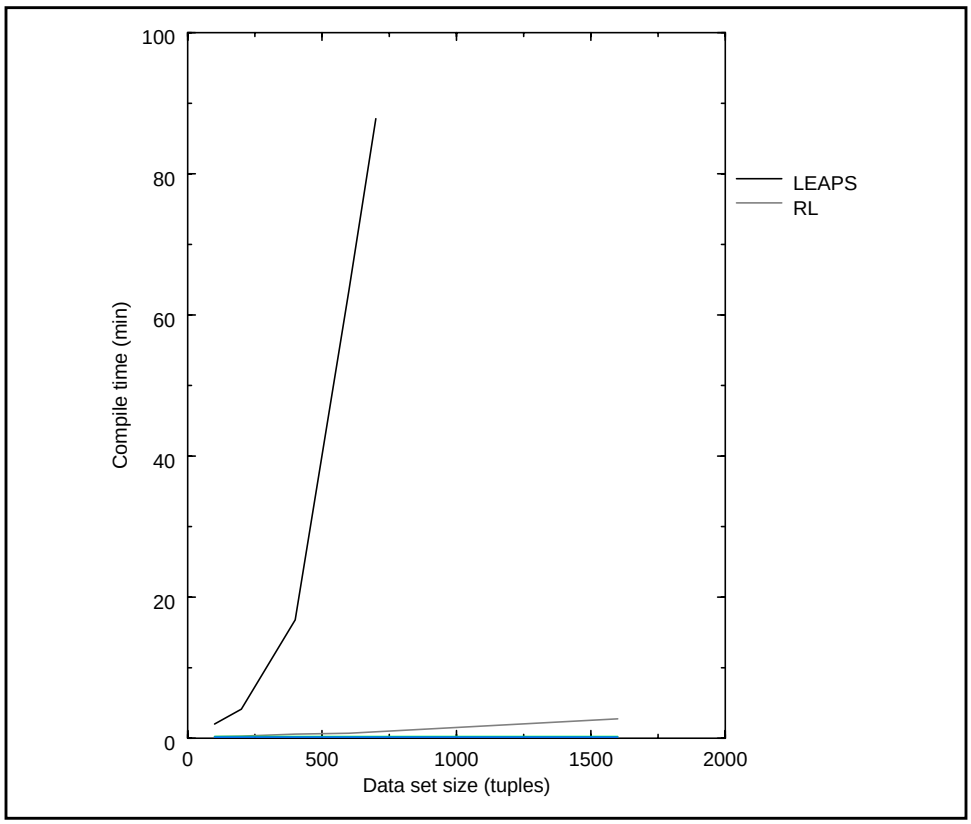

Figure 5.13: Compile times of jig25.ops

The following performance graph, therefore, shows four different traces. For each system (LEAPS, Pred) there is a trace for data from a data file and one for data located in the program[10].

9. They would prefer to have the data in the rules (for faster execution speed), but are forced, for "large" data sets to place the data in an input data file, which must be read at run-time.

10. The lower two traces are from Predator, and the lower trace for both Predator and LEAPS are for the executions where data was stored statically in the program.

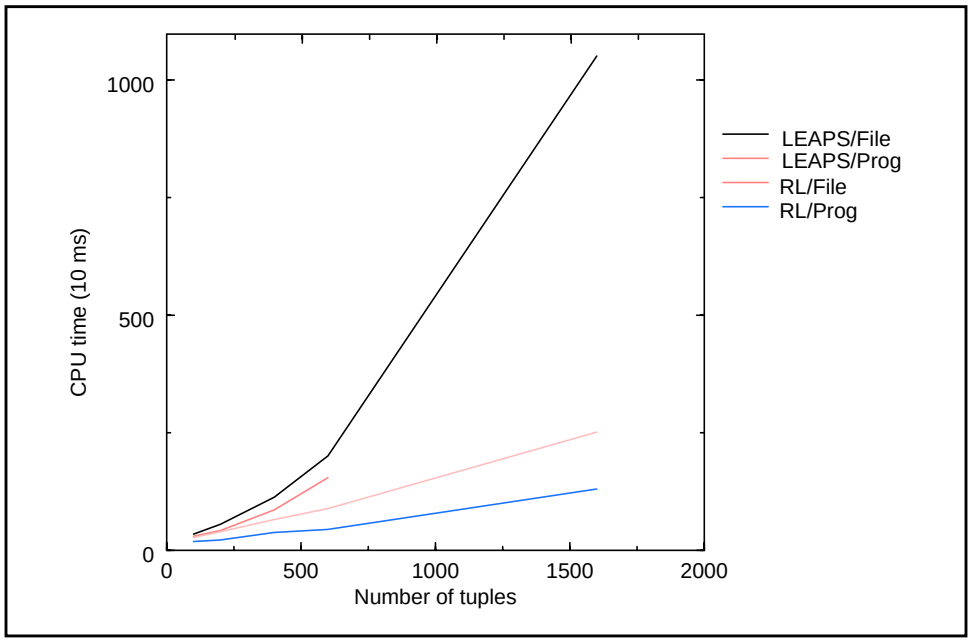

Figure 5.14: Execution speed of jig25.ops

This test case shows an even larger performance differential between Predator and LEAPS than in previous test cases. In addition to the performance advantages mentioned in previous sections, there is another reason why the RL version of jig25.ops performs better than LEAPS. In both systems, predicate index nodes are linked in the following way: Each predicate index node that refers to a given WME is linked in a chain, with a head pointer stored in the WME. This provides quick reference to all of the predicate index nodes for the WME (if the WME is deleted or updated). If, in a given rule set, there are many different predicates (from many different rules) which are the same predicate, LEAPS will create one predicate index per predicate. To illustrate:

```
(p rule_1
 (a ^b > 5)
-->
 (action1)
)

(p rule_2
 (a ^b > 5)
 (other CEs)
-->
 (action2)
)
```

LEAPS would generate two different predicate index chains for the predicate shown here. As each WME is inserted in the database, not only does it potentially have to be added to two different predicate index chains, but those predicate index nodes also have to be interlinked. This is quite inefficient, particularly as the number of predicate indexes and number of WMEs grow large.

RL, by contrast, can “share” the predicate indexes. It will generate only one predicate index for the example above. In a given rule set there can be a great many condition elements which are exactly the same. Thus, the efficiency gain can be dramatic.

This is yet another case where the high-level abstractions of Predator allow for features to be added simply to the system. The LEAPS programmers are aware of this inefficiency. It's just too difficult (currently) to rewrite the code to remove the problem. With Predator, the change can be effected with little trouble.

The final negation test case is called waltz.ops, and is very large (4 containers and 33 rules). waltz.ops is a PS to solve a complicated two-dimensional puzzle. The LEAPS team uses waltz.ops as a standard of large production systems. This was our last and most complicated test case to emulate. Not only did we emulate waltz.ops, but the RL version outperforms the LEAPS version (for all the reasons described earlier).

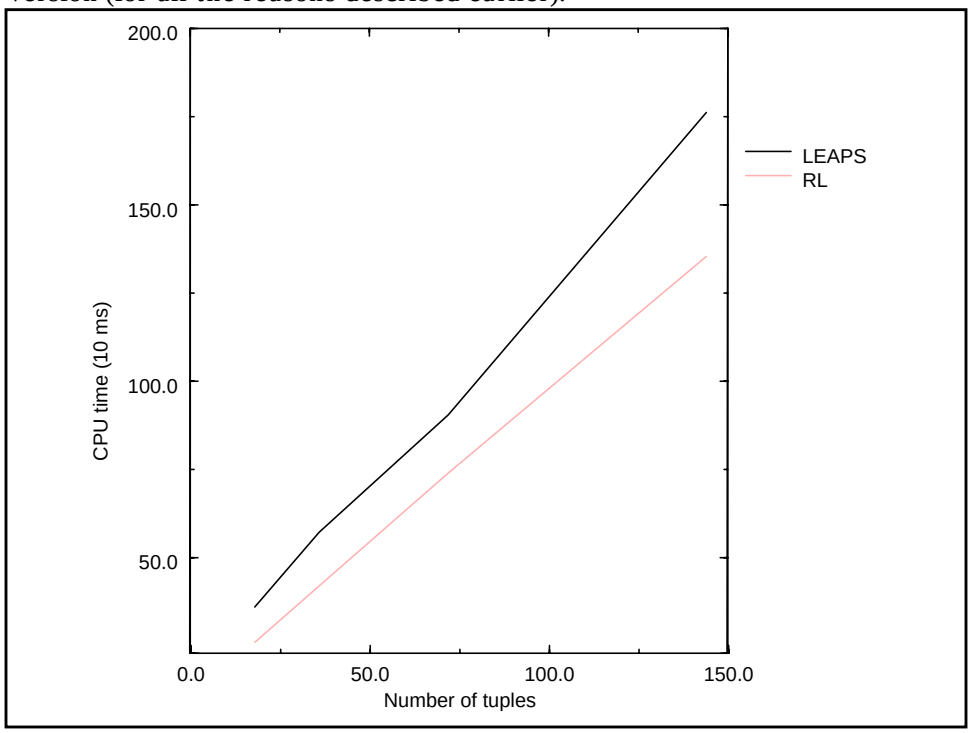

Figure 5.15: Execution speed of waltz.ops

5.4.7 Persistent production systems

All of the test cases described above utilize main-memory data structures. There is no reason, however, that both the production system's data and/or the control structures must reside in main memory.

Adding persistency to production systems produces two major benefits:

1. Significantly larger data sets can be processed in this manner.
2. A production system can be stopped in the middle of processing, and restarted at a later time.

A persistent version of LEAPS, called DATEX [Bra93a, Bra93b], already exists. DATEX programs are used to handle very large data sets, but cannot be stopped and restarted.

A memory-mapped version of persistency was added to Predator as a layer. Due to the data structure independent nature of Predator, any of the production systems generated by RL can be made persistent. Several of the test cases were made persistent. The following is a type equation from the persistent RL version of waltz.ops.

```
SCHEMA stage_struct ON ELEMENT stage =
       dlist[timest[persist[waltz.per,
                            20000],
                    GLOBAL]];
```

In addition, the control structures (including the wait-list and wait-stack) were also declared to be persistent. A test version of *waltz.ops* was created that would execute 10 production cycles each time it was executed, with the intermediate results stored in the persistent data structure. It reached the same correct fix point as the memory-resident version.

Three of the main-memory test cases were compiled and evaluated as persistent productions systems for both DATEX and RL. The results of those tests is shown inTable 5.8.

Table 5.8: Execution times for persistent production systems

Production system	Input set size	Persistent RL time (sec)	% of main-memory RL	DATEX time (sec)	% of main memory LEAPS
big-join	1000	4.3	108%	506.3	7110%
jig25 (prog)	100	0.2	111%	21.0	6960%
triples	35	1.5	111%	178.7	8123%

Adding persistency to the RL driver caused it to execute approximately 10% slower than the main memory version of the same driver. DATEX, on the other hand, suffered a performance penalty that caused it to run approximately 70-80 *times* slower than the LEAPS version for the same production system. While this number is huge in comparison to the RL equivalent, it can mostly be dismissed. DATEX utilizes persistency via the JUPITER module of the GENESIS database system [Roy91]. JUPITER is a general purpose system, which is not particularly efficient. Thus, comparing it performance-wise to a simple memory-map is not particularly meaningful.

It is interesting, however, to see how long it took to add persistency to both LEAPS and RL. The LEAPS programmer estimates that it required approximately two months of programmer time to add and debug JUPITER to LEAPS to create DATEX. Persistency in RL, however, consisted of one new layer to make calls into the memory-map library module, which required less than two days to program and debug.[11] Again, the layered architecture of NPTs yielded improvements in programmer productivity.

5.4.8 Future work

All of the test cases reported in previous sections consisted of two sections:

1. The RL main driver, which is common to all the test cases.
2. A test-case specific section of code containing the condition elements and actions for the rule set.

For all of the production systems described in this chapter, the specific portion was written by hand. Even though it was custom-written, the code is extremely regular. It was written in such a manner that it could be the result of a mechanical translation of the original OPS5 rule set.

An automated RL generator is currently under construction. When completed, it will read OPS5 source files and produce Predator code. That Predator code can then be precompiled, combined with the RL driver module, compiled, and linked to produce an executable. When

11. This time also includes the time required to add stop and start functionality, which DATEX cannot do.

finished, the automated RL precompiler will be able to generate all of the test cases which can currently be generated by the LEAPS compiler.

5.4.9 RL conclusions

The LEAPS/OPS investigation has proven extremely helpful. It has demonstrated that many of the advantages of the NPT model do extend to large, complex program development. Specifically, we were able to emulate all of the LEAPS test cases while retaining the following advantages:

1. Performance. The RL programs were all faster than their LEAPS counterparts. This was due to design decisions, as well as to tighter, optimized code generated by Predator.

2. Programmer productivity. The RL compiler, when completed, will have taken a lot less time to write and debug than did LEAPS. Much of this is due to the fact that the LEAPS programmers wrote many thousands of lines of data structure code, which took significant time to write and debug. The equivalent RL code was automatically generated, and did not need to be debugged.

3. Extensibility. RL contains some features (such as persistent start and stop, multiple link implementations, ability for input data to be located in the rules themselves, sharing of predicate indexes, and automatic static-typing) not found in LEAPS. This is due to the simpler high-level design of the RL code. In addition, new features should be easier to add to RL in the future.

5.5 Validation conclusions

When this research was begun, we hoped to design and construct a prototype system which would make programming data structures much easier. We also hoped that the efficiency of the generated code would be “good”. At the time we defined “good” as within 10% of hand written and optimized code.

We designed a series of experiments, ranging in size from function, to component, to system level. The experiments were designed to measure programmer productivity, ease of programming, and performance. Also, the tests were designed to compare this approach against more traditional methods of coding.

The results obtained from these experiments have been better than we had hoped for. While we had expected to see productivity gains over traditional methods, we had not expected to see the types of performance gains we have seen from our Predator prototype[12].

We feel that the range of results obtained from these experiments clearly shows that the original design objectives for this research have been met:

- In all of the experiments described, writing Predator code has taken less time than writing the code either by hand or by using presupplied modules (queue, Booch, libg++). Predator appears to improve programmer productivity.
- Programmers have used data structures about which they knew very little. Both in the queue experiment and the RL experiments, programmers used data structures without knowing the inner workings of the data structures involved.
- Programs were debugged at the algorithmic level, without having to observe the inner workings of the data structures.
- Programmers were able to substitute data structure implementations (in the spelling checker, quick sort, and RL experiments) without having to change the actual application code. Only the data declaration was changed.
- Programmers were able to add additional features (segmentation, index sharing, persistent start and stop etc...) easily, due to the high-level algorithmic code of Predator.
- All of the generated code produced performance that was within 10% of hand written code. In almost all cases, the Predator code was superior to that either written by hand or supplied by previously written data structure modules.

12. The expectation was that better performance would depend on more domain-specific code optimizations being added to the prototype.

Chapter 6

Related work

We mentioned several approaches in Chapter 1 which have been explored to simplify the burden of data structure programming. In this chapter, we highlight representative systems of those approaches, discuss their strengths, and explain how they differ from our work. In addition, we examine the problems of general software reuse [Big89a, Big89b]. There are many researchers in software reuse who are not addressing data structure reuse in particular, but whose insights are relevant to our work. Finally, we mention several projects which are related to ours in that they use the GenVoca domain modelling concepts.

6.1 Software reuse

A comprehensive exploration of software reuse was made by Krueger [Kru92]. He proposes a taxonomy of five key attributes which are claimed to be common to all software reuse approaches:

1. The *artifacts* the approach uses.
2. How the artifacts are *abstracted* into usable features.
3. *Selection* defines how the software reuser can find usable artifacts to reuse.
4. *Specialization* refers to the process of refining a generalized artifact into a more specific one via parameterization and feature selection. Note that this is a more restrictive form of specialization than we defined in Chapter 3.
5. *Integration* describes the framework that the reuse technology uses for combining the selected and specialized artifacts into an existing application or framework.

He then defines eight approaches to software reuse: high-level languages (not generally considered a reuse technique - but Krueger makes a good argument for considering it so), design

and code scavenging, source-code components, software schemas, application generators, very high-level languages, transformation systems, and software architectures. Each approach is illustrated by an exemplar system.

Krueger defines the term *cognitive distance* to be a rough, intuitive gauge to compare different reuse abstractions in terms of the intellectual effort required to use the abstractions. He describes three rough metrics to use in measuring cognitive distance:

1. Using abstractions (both fixed and variable) that are both succinct and expressive.
2. Maximizing the hidden part of the abstraction.
3. Automatically mapping from abstraction to realization (implementation).

These metrics are both intuitive and easily measured. We feel that NPTs measure up well in a cognitive distance measuring. To a large degree, e this is borne out by Krueger's measurements which, overall, rank application generators, transformational systems, and software architectures as among the best approaches for minimizing cognitive distance.

A slightly different, but also useful view of reuse can be found in [Raj89]. Raj concentrates on object-oriented languages and software reuse. His thesis is that inheritance-based schemes of software reuse are often inadequate. He advocates, instead, a compositional method of component creation for such systems.

He begins by developing metrics to gauge the effectiveness of object-oriented languages in terms of methods, classes, and applications. He then applies these metrics to several object-oriented systems. He notes that most inheritance hierarchies tend to be more broad than they are deep (thus questioning the amount of inheritance truly necessary), and notes several conflicts that inheritance has with conformance and locality.

He proposes a compositional model and prototype (the Jade programming language) based on the Emerald programming language [Bla86, Bla87]. His language advocates a Lego-block composition similar to that of GenVoca, by extending Emerald with constructs which allow for the declaration of compositional names and dependencies, and by including of parts of other components previously defined. Jade also provides an environment for component reuse via a set of tools, such as component browsers.

Another point of similarity is that Raj discusses black-box and white-box reuse, which are analogous to opaque and transparent components. Much as we note (see Section 3.4.1) the need for *semi-transparent* access to components, Raj describes *grey-box* reuse, in which the

internals of the component are exposed, but not modified. This is somewhat more open than our semi-transparent approach, mainly because his compositional model allows different components to have different interfaces. Thus, it is important to be able to view the internals of components to determine the proper interconnections.

Raj also argues that compositional methods are often superior to object-oriented ones. He investigates four large (> 30,000 lines of code) object-oriented systems, and measures the amount of class reuse in the hierarchy. He notes that each of the systems reviewed have very little reuse of classes. The inheritance tree for each project is very broad and shallow. This runs counter to the objectives of object-oriented programming, in which deep class trees are a goal. His arguments support our NPT compositional view of the world, in which components share a similar interface and are composed to create new components.

As with many of the other systems reviewed in this chapter, Jade is primarily general-purpose. Thus, it differs from our work, which is targeted at a specific domain. Further, Raj provides a compositional mechanism for defining components with different interfaces and a system for implementing the compositions. Our work differs in that it deals with components that share a common interface, and we provide only a static mechanism (vertical parameterization) for composition. We do not support dynamic realm creation.

6.2 Software component libraries

There are many software component libraries currently available, and creating a good component library is a complex task [Kic92]. A large subset of available libraries implement data structure components. In this section we will discuss two of them.

The Booch components are a set of monolithic data structure components. The original set of components [Boo87] were written in Ada. There are a large number of components in the library (more than 400), and the interfaces of these components are mature, high-level, and well-defined. Booch has defined an interface that includes such items as constructors, iterators, selectors, and exceptions. He also has defined a set of *tools*, such as sorting, character manipulations, numerics, and list operations. These tools are implemented across the family of components. He also has implemented components for concurrent and persistent data structures.

Booch components are also available for C++ [Boo90]. The conversion to C++ takes advantage of object-oriented features of C++, and allowed Booch to dramatically reduce the size of

the components, while retaining the same interface and overall performance as the Ada version.

The Booch components possess many of the limitations that we outlined in Chapter 2. Specifically, the component interfaces are not consistent, the Booch model does not allow for arbitrary compositions, the performance (as seen in Section 5.3) is not competitive with other TPT libraries, and the library is not scalable [Bat93a].

libg++ is another C++-based component library [Lea88]. libg++ is a collection of classes and support tools that is part of the GNU[1] tools distribution. As such, it is freely available, and has been heavily used and tested. Lea notes the distinctions of libraries (such as OOPS [Gor87]) which are based on a single, consistent hierarchy tree of classes; and libraries such as libg++, which are based on a forest of smaller trees of related classes. He discusses the issues of usability versus performance, which are typical trade-offs for generality versus specificity. libg++ contains a fairly complete set of data structure container classes and operations to access them. Again, the interface is not too dissimilar to that of the Booch components. This leads to many of the same strengths and weaknesses found in the Booch library.

6.3 Transformation systems

Another class of systems mentioned previously are transformation systems. There are many examples of transformation systems both within and without the data structure domain. A survey of several early transformation research efforts can be found in [Par83]. Partsch and Seinbrüggen describe many relevent research efforts, such as SETL [Sch79], Draco [Nei80, Nei84], and the Programmer's Apprentice [Ric79, Ric90]. Gries [Gri90] also presents a good formal introduction to transforms as a language construct.

The Interface Description Language (IDL) [Sno89] allows users to define a system as a pipeline of tools (or processes), each of which performs a specific task. The interface between these tasks is then specified in a rigorous way. The IDL compiler creates input and output routines for the different tasks. The generated code can be instantiated to utilize different data structures and modules to transform the generated code to a more efficient form. In addition, IDL provides an *assertion* capability which allows data invariants to be declared and verified.

---

1. GNU stands for "Gnu's not unix", and is a product of the Free Software Foundation.

IDL provides a number of benefits, including the formality of the specification, data structure and source language independence, and maintainability. IDL-generated routines can be moved among different applications, compilers and systems with little effort. The assertion capability, coupled with the separation of the IDL protocols from the task's implementation, allows for simpler maintenance and evolution.

IDL does have limitations. IDL programs are inherently sequential and are used primarily for pipelined software architectures. The assertion capability is also restricted to the IDL I/O data types; data cannot be asserted while it is being manipulated by a task, only after it has been output from one stage and before it is input to the next.

GLISP [Nov83, Nov92] is a transformation system which allows generic procedures to be specialized through a concept known as *view clusters*. View clusters are similar to NPT type equation instantiations, except that view clusters are not layered. However, GLISP has the additional advantage that view clusters allow fields in a data element to be shared among different view clusters, and thus used for different purposes within different views. Also, GLISP supports both in-lining and optimizations. Further, GLISP provides the capability for operations to be *side-effect* free, which can improve program correctness. GLISP is unlike NPTs in that it does not support consistent, layered interfaces or flattened data structures.

Another LISP-based data structure transformation system is AP5 [Coh89, Coh93]. AP5 is a common-lisp derived system which augments the standard lisp syntax with constructs which are transformed into calls in a run-time library. Like our system, AP5 is relational and uses a high-level interface. AP5 extends the model by allowing annotations to be made to transform data and improve efficiency. AP5 also supports invariants through *consistency rules*, which are stated invariants coupled with repair actions to be taken if the invariant is violated.

While the basis of AP5 is relational, the syntax is lisp-based and does not resemble classical relational database syntax [Kor91]. AP5 relies on the concept of the *fact*, which is a LISP list, where the car of the list is the relation name, and the cdr of the list is the data for the tuple. Operations on the facts can then be expressed as a different list, where the car of the list is the operation to perform on the fact, and the cdr is the fact itself. AP5 also supports the notion of the *annotation* to declare the relation (container) implementation.

While AP5 has an interface which is consistent and fairly closely resembles the NPT interface, it, too, suffers from the lack of component layering. Thus, we feel there are limitations on scalability, complex compositions, and type transformations that it can perform.

6.4 Other approaches

The fields of software reuse and data structure programming are vast. This section briefly describes several other approaches researchers have investigated in these areas.

Module Interconnection Languages [Pri86, Raj89] are another approach to the data structure programming problem. MILs concentrate on defining the interfaces of software components, how to import and export the interfaces, and how to interconnect the components together to form a new, higher-level component. This is useful in creating complex data structures, and offers opportunities to build more complicated data structure components from other smaller, interconnected components.

There are many *object-oriented languages* which offer useful features for NPT systems. Object-oriented programming is currently becoming a method of choice in data structure programming. Object-orientation offers many advantages; it provides a consistent abstraction (the object), allows for reuse via inheritance, and is a well-understood and appreciated paradigm. Many object-oriented systems and languages have been used for data structure programming and libraries. Some of these are Smalltalk [Gol83], C++ [Str91], Modula-3 [Har92], SELF [Cha89], Eiffel [Mey88], and Meldc [Pop91]. All of these suffer from the limitation that they possess no native knowledge about specific problem domains. Thus, they cannot natively perform domain-specific optimizations. We have also detailed, in Chapter 2, the compositional difficulties object-oriented approaches possess when dealing with complex compositions. In recognition of these lacks, we are investigating a system which combines object-oriented programming and our NPTs [Sin93].

Many of the concepts and techniques we use in this research have also appeared in other research projects. *Parameterization* is one example. The NPT model is based heavily on parameterization. Goguen has formalized aspects of component design in a model called parameterized programming [Gog86]. This model identifies two kinds of parameters: vertical parameters (which specify lower layer components) and horizontal parameters (which correspond to type and constant values). Both horizontal and vertical parameterization are used in NPTs.

Software templates for NPTs were described in Section 4.1.2. [Vol85] describes a methodology for software templates. He recognizes the importance of separating algorithms from the implementation of the data structures. However, without layering (i.e. vertical parameterization), we find templates insufficient to model the full range of data structures available with NPTs.

6.5 GenVoca systems

Our NPT work is based on the GenVoca domain modelling approach [Bat92b]. The concepts for GenVoca are based on two independent projects; Genesis [Bat88, Bat90], which is a generator of database systems, and Avoca [Oma90], which is a generator of network protocols. Despite the radically different domains for the two projects, it was observed that they approached the generation of software systems in a similar manner. A number of subsequent projects have embraced the GenVoca approach. In the domain of data structures, both this work (Predator) and Predator-2 (P2) [Bat93a] address the generation of efficient data structure code. P++ [Sin93] provides an extension of the C++ language which allows definition and combination of realms and components, based on the lessons learned from Predator. [Abb92] describes a protocol customization compiler which is based on the Avoca project and bears many similarities to the Predator system. ADAGE [Cog93] is a GenVoca-based system in the domain of avionics software. Finally, FICUS [Hei90, Hei91] provides a layered approach for customizing file systems which is based on concepts similar to those in GenVoca.

Chapter 7

Evaluation

This chapter presents an assessment of our research. As documented in Chapter 5, we feel that we have achieved our major goals. However, in looking back at our efforts over the past 18 months, we feel that it is appropriate to note the limitations of our effort, as well as choices we made which have not turned out for the best, and which we would change if we were to start over. The evaluation is organized into two sections, one dealing with conceptual limitations of our work and the other dealing with implementation limitations of our work. It should be mentioned that corrections for most of the limitations we discovered during our research have been incorporated into our next generation tool, P2.

7.1 Conceptual limitations

In this section we discuss some of the conceptual problems we see with our NPT model for data structures.

Physical pointers within cursors. A feature of our NPT design was to store all element references (within cursors) as a single untyped pointer (a C-language **`void *`** type). Further, we imposed the restriction that element references within cursors could not be augmented by the components. For example, while a **`PREDIND`** component might wish to augment a cursor definition with an extra pointer, our model did not allow this. While limiting cursor definitions to single pointers has an advantage of simplicity, it also has the drawback that certain data structures are difficult to represent. Some structures, such as B+ trees, do not assign fixed addresses to records. Record pointers for such structures are logical (i.e., key-valued) rather than physical (i.e., addresses). Consequently, our choice of cursor implementation makes it difficult for us to implement B+ tree-like data structures.

We implemented a layer which illustrates the problem of physical pointer references. The **PINDEX** layer performs predicate-based indexing, with a list of elements which satisfy the predicate being stored externally to the elements. Figure 7.1 depicts a two populated **PINDEX** structures with the following declarations:

```
SCHEMA test_schema ON ELEMENT customer =
       pindex[pindex[malloc[],
                     age == 30],
              strcmp(state, "TX") == 0];
```

To build and maintain layers such as **PINDEX** or **BTREE** requires that we augment the cursor declaration. Whereas it is normally possible to represent a cursor's current position with a simple physical pointer, it now becomes necessary to point to the external **PINDEX** node, rather than the element itself. This allows for operations such as **ADVANCE** and **REVERSE** to be performed properly. There are other possibilities available in which hybrid pointers (to both the element and the external structure) are possible. Additional work is required, however, to fully understand the performance trade-offs (as well as the needed additional language support) for the various options.

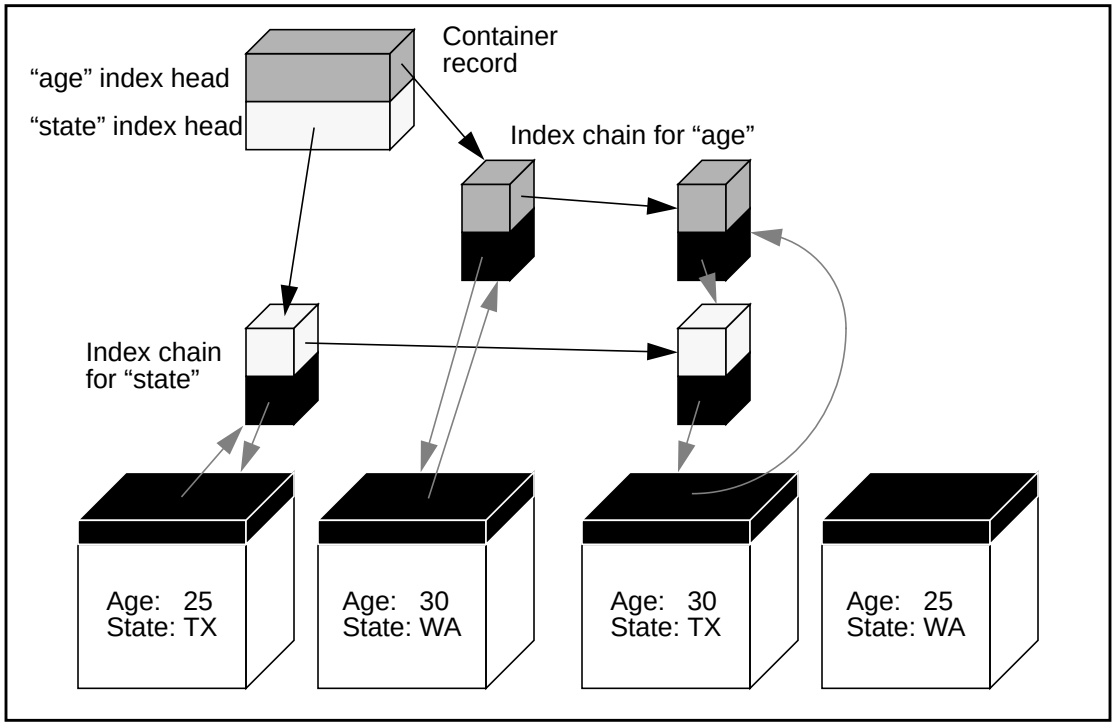

Figure 7.1: Predicate indexing with **PINDEX**

Inter-layer knowledge. Our current NPT model specifies that layer-specific knowledge should not be passed among layers. More formally, if two layers, **L** and **M**, are declared in type equation **T**, with **L** being declared above (before) **M**, then **L** can have no knowledge of **M**'s type or implementation. **M** can possess knowledge about **L**, since the original element's type will have been modified by **L** before **M** receives the type as input.

The realization of our model, however, did not maintain a clean abstraction in all cases. There were certain layers which we found difficult to implement without breaking layer encapsulation. A good example of this was segmentation. For efficient implementation, we found that some of the details for segmentation were pushed up into the compiler itself. This helped improve the computation speed of compilation, and made the layer easier to write.

We broke encapsulation on only a few layers. Nevertheless, a more formal handling of this is required. Either the layers all need to be written within our stated abstraction, or we need to declare formally when and how it is possible to break the barrier. Failing that, future layer writers might be tempted to break the barrier in various ways, which could quickly get out of hand.

No "order-by" in cursors. Our current cursor declaration syntax allows for an optional **WHERE** clause, but does not allow an **ORDER BY** clause. This was an oversight, which has become an increasingly larger problem as time has gone by. Although it is possible to emulate the **ORDER BY** clause by manually placing ordered layers in the type equations, using those layers for scanning, this mechanism is cumbersome.

Supporting this feature explicitly is more difficult than simply externalizing the process described in the previous paragraph. Parsing the syntax and managing the symbol table for **ORDER BY** are not too difficult. The major hurdle with this feature is as follows: It is possible, when one can explicitly declare an **ORDER BY** clause, for an application writer to require that retrieved elements be ordered by a particular field when the container's type equation does not specify that particular ordering for any of its layers. The following example illustrates the problem:

```
SCHEMA order_by_schema ON ELEMENT customer =
       dlist[malloc[], age];
order_by_schema order_by_container;
CURSOR order_curs ON order_by_container ORDER BY l_name;
```

There is simply no knowledge present in the container definition as to how to order the container properly for the cursor's needs. There are several solutions to this problem:

1. Require the compiler to generate the (database-like) code necessary to order the records whenever a read access is made to the ordered cursor. This has the problem that the resulting code would be quite slow.

2. Have the compiler automatically insert an ordering layer (such as an ordered list or ordered tree) in the type equation to handle the cursor's needs. This solution would add overhead to operations such as **`INSERT`** and **`DELETE`** because there would be more layers than the programmer specified. This maintains the ordering required by the **`ORDER BY`** clause at all times.

3. Not allow this feature. It would be fairly easy to have the compiler report this mismatch as an error. In this case, the application programmer could either add the ordering layer to the type equation, or change the definition of the cursor.

Of the three solutions presented above, we prefer the second. It is fairly simple to implement (we have written similar support for link layers), it simplifies the work of the programmer (over the third solution), and it will be more efficient in program speed (in general) than the first solution. The drawback to this solution, though, is that the compiler does not understand the programmer's algorithm, and may be adding a layer which will be very inefficient. We propose that option 3 above also be implemented, so that the compiler will report a warning (not an error) that it is augmenting the type equation for the programmer. This notifies the programmer of the mismatch, and allows them to further investigate, if desired.

Layer-specific primitive functions. In Section 3.3.4, we described a high-level interface that we believe is sufficient for most common data structure programming tasks. There are layers, however, which require that additional primitive functions be provided. A good example of this is the **`size`** layer, which adds an integer field to the container record. The integer field records the current size of the container (i.e. number of elements stored in the container). None of the predefined functions can handle reporting on the new field that **`size`** adds. Other layers, such as **`action`** and **`timestamp`**, also add new functions.

Our current solution to this problem is to add a primitive function to every layer for the new layer's function. So, in our example, a new **`get_size`** function was added to each existing layer. The only **`get_size`** function that generates code is the one in the **`size`** layer. All of the other layers simply ignore the function, and pass it to the next lower layer.

As the number of custom layers increases, this solution suffers from the same scalability problems we noted with TPTs. Placing a large number of empty functions in existing modules each time a custom layer is added is not a viable or scalable solution.

We believe that the solution rests with augmenting both the layer definition we use and the dispatch table used by the code generator to traverse the type equation tree for a primitive function. Essentially, we propose that each layer specify (as part of its definition) any additional functions it requires. When the NPT compiler starts, it would load the names of these functions into a special portion of the symbol table. When a special function name is tokenized, the compiler would call on a new part of the dispatch table routine, which would do nothing more than traverse the type equation tree until it arrives at the layer that provides the function. It would call on that function, and add the resulting code to the output stream. Thus, existing components would not need to ever have any knowledge of these new functions; as the compiler and the layer would be responsible for generating the correct behavior. This dynamic extension of the realm interface is also the solution used in similar system generators for other domains [Hut91, Hei91, Pag90].

Self-referential links. In Chapter 5, we mentioned the lack of self-referential links. That is, the composite cursor declaration syntax we defined did not allow for more than one reference to be made to a particular cursor/container pair. Because of this, several of our tests could not be performed using both nested loop and pointer-based link algorithms. In addition, some of the code we wrote for the OPS5 test cases was expressed in terms of link cursors and container cursors, instead of the higher-level composite cursor constructs. Neither of these were serious problems for our work. However, they both required us to spend additional time and effort to duplicate what should have been trivial operations. This experience ran contrary to our goals for our NPT work. All of this effort could have been avoided if we had seen the need for self-referential links before we began their design and implementation.

Adding self-referential links to our model is not difficult. Consider the following augmented declaration for composite cursors:

```
COMPCURS <cc_name> USING {<c1> <cont1>}+
        WHERE <cc_pred1> (&& <cc_pred2>)*;
```

The composite cursor declaration is similar to that presented in Chapter 3. The only difference is that each **<cc_pred>** clause which is a link reference is augmented with the container cursor(s) that operate on it. For example:

```
link2(p_curs, c_curs)
```

refers to a link in which **p_curs** and **c_curs** are the parent and child cursors to use when instantiating the link. Thus, the ambiguity caused by having more than one container cursor reference the same container is removed. The compiler then possesses enough information to process the composite cursor. The following code demonstrates this new feature:

```
LINK manages ON ONE employee TO ONE dept
             WHERE employee.emp_num == dept.manager;
LINK works_in ON MANY employee TO ONE dept
             WHERE employee.dept_num == dept.dept_num;

COMPCURS cc USING e1 employee e2 employee d1 dept
          WHERE manages(e1, d1) && works_in(e2, d1);
```

This composite cursor will find all employees who work in a given department, along with their manager. Even though more than one container cursor refers to the employee container, the compiler can properly resolve the references.

7.2 Implementation limitations

The following is a list of the major limitations of our current prototype. Because Predator is a prototype, it was designed in an incremental fashion. Many of its features simply *grew* over time. This section describes those features that we feel would be crucial to design better in a future version of Predator. In fact, many of them have been improved in P2.

No proper scanner/parser/tokenizer. This is a major problem with the current implementation of Predator. The initial version of Predator was intended simply as a throwaway technology demonstration. It was not intended to be the base for our prototype. The scanner/parser/tokenizer was custom-made, and only intended to operate on a very limited grammar. As time passed, we simply kept adding to the grammar that our parser recognized. The enlarged grammar revealed holes in our parser/tokenizer, which we patched as well as we could.

Our prototype is almost at the breaking point. There are far too many patches and workarounds for this system. It is unlikely that we could add many new features before the parser would become totally unmanageable.

We believe that the correct solution is to use tools such as **`lex`** and **`yacc`** to generate these sub-systems. By starting with a proper C grammar (which is commonly available), we can then augment the grammar with our NPT constructs. This is exactly what has been done with both P2 and P++. In hindsight, we would have saved ourselves a great deal of time if we had taken the time to backtrack with this approach long ago.

Limited dynamics. Our NPT design does not differentiate between static and dynamic entities (containers and cursors). As we built the prototype compiler, we discovered that all of our initial test cases only required static entities; as a result, we concentrated on statics. Upon reflection, we realize that this was a mistake. While there is no conceptual difference between our static and dynamic entities, there is a significant difference in their implementation. Our compiler implementation would need to be improved in two areas to allow for dynamic entities. The first is that we need to be able to parse and type check constructs such as type casts of C variables to Predator types. This should be fairly simple to accomplish with the improved parser described above.

The second item that needs to be modified to support dynamics is better run-time support. There are many tasks having to do with initialization, allocation, and optimization which are currently handled by the compiler (for static entities). The data structures to handle these features exist only at compile-time. For dynamics, many of these data structures and algorithms need to be brought forward to the run-time code. This is not difficult, but the main concern would be to design these algorithms to be efficient in time. Currently, we have emphasized correctness over efficiency in these operations, since speed is not a concern in the compilation stage. That would change for run-time processing.

No multiple file support. All of the code for a Predator program must currently be located in one source file. This restriction is due to the fact that we have provided no support for including secondary source files as part of the compilation process. We proposed a **`#predator_include`** directive for our programs, similar to the C preprocessor, but it has not been implemented. The reason for this is that our testing has involved relatively small source files. Also, Predator is not mature enough that we have built large libraries of support functions which we would want to include into our source files. Implementing this function is not difficult, and has been done in P2. In P2, the C pre-processor is run before the Predator compiler, and thus the secondary source files are expanded and merged together into one source file. This was not possible with the original Predator, due to the fragile nature of our parser (discussed earlier).

Simplify layer writing. Predator layers are fairly simple to write. Nevertheless, they are more complicated than they ultimately need to be. As the number of layers increased, the complexity of each layer also increased (slightly). This was due to the fact that we had no proper mechanism in place to handle layer-specific functions (see above). We implemented a tool called **`layergen`** to help with this task. **`layergen`** is a program which creates the constants, variables and type equation parsing routines for each layer. It is called as part of the compiler's make file, and aids in adding new layers. The idea is that the writer of new layers should not have to know anything about the internals of compiler.

Helping the layer author write the layer module is a different task. Much of the code within each layer is the same as all other layers. Code scavenging among layers helps some, but a layer generation tool seems to be the proper direction. Based on this, P2 has included the **`xp`** tool. **`xp`** takes a series of layer specification files (in a standardized format) and generates the layer modules. In the future it should be possible to use such a tool to help standardize the identifier names used in the layer modules, as an aid to the optimizer module.

Generated code size. In the current Predator environment, we generate code via macro expansion and optimization. That is, for a given primitive function (e.g. **INSERT**) we generate a code fragment based on both the type equation for the container and the predicate of the cursor. It is common in most application programs that there will be more than one call to a given primitive function for a given container/cursor. For example, suppose a program has a container named *employees*, and a cursor (defined on *employees*) named *e*. Further suppose that there are multiple locations in the program where the primitive function **`DEL(e)`** is called. Predator will generate ***exactly*** the same code fragment for each instance of the primitive function. When one considers the number of repeated function calls in a program of even moderate complexity, it is easy to see that the size of the generated code can be a problem.

There is a solution. It is possible to generate a new function for each primitive function/container/cursor combination. Then, each call, in the source code, to a primitive function is replaced by a simple function call in the generated code. There are two problems with this approach. The first is that this solution creates additional run-time overhead for the new function calls. The other problem is that not all of the primitive functions are used in a given program. Since macro expansion only creates code for those functions which are used, it is possible for the new program to be even larger than the one it is replacing. Therefore, care must be taken to generate new functions only for the primitive functions which are used in a given application program.

Deletion semantics. We took a very simple view of deletion in our prototype system. When an element is deleted, cursor stability is ignored. That is, if there are any cursors which are currently referencing the deleted element, it is up to the cursor (and the routines that use it) to ensure that it be reset to a non-deleted value. With these semantics, it is possible for a programmer to access memory illegally if they do not first check the validity of the cursor's value. This restriction was not a problem for the sample programs we wrote for this research; but, we realize that in many other environments, such as multi-threaded programs, this restriction would be a real burden.

There are many possible solutions to this problem. The most obvious of them is to augment the generated code for the **DELETE** function so that it is responsible for setting all cursors (not just the one referred to in the **DELETE** function) to the *next* elements in their scanning paths (or to set their status flags to EOR if there are no more elements). This behavior could be easily encapsulated in the **DELFLAG** layer. As with many of the other limitations discussed in this chapter, more formalism is required to define the desired semantics. Work in this area is currently being integrated into the P2 prototype.

Chapter 8

Conclusions

8.1 Summary

Data structure programming can be quite time and labor intensive. We have found that many current solutions to the problem (e.g. TPTs) are insufficient to adequately handle issues such as specialization, complex compositions, ad hoc interfaces, type transformations, evolution, scalability, and code efficiency.

We have proposed a concept called non-traditional parameterized types (NPTs) which resembles concepts and features from a variety of disciplines as the basis for practical software generators for data structures. We have described the features of NPTs and shown how they overcome a number of serious limitations of traditional parameterized types (TPTs).

A prototype generator named Predator was constructed. Predator is a system generator/ data structure precompiler which is able to construct customized data structures by composing primitive data structure components. Predator takes, as input, C-language programs which have been augmented with Predator declarations and primitive functions, and generates standard C-language source which can be compiled and linked with standard tools. To date, we have built 17 different components, which implement data structures such as arrays, lists, trees, predicate-based indexing, timestamping, and segmentation. Predator generates static and dynamic allocated structures, as well as transient and persistent structures. In essence, Predator elevates data structure programming to the relational level where different implementations of relations can be selected according to performance requirements.

We validated both our model and implementation with a series of experiments. The experiments measured both programmer productivity and code efficiency. They ranged from small programming tasks written by professional and non-professional programmers, to module sized algorithms, to emulating a production system compiler. In each case, we noted the

advantages and disadvantages of our approach. We have compared Predator against both hand-written, optimized code and pre-written data structure modules. Finally, we benchmarked the performance and programmer productivity of our compiler against two commercially available component libraries across four common data structures. Overall, we found that Predator noticeably enhanced data structure programming productivity, as it eliminated the mundane and complex details of writing data structure code. Moreover, the performance of Predator-generated code was comparable, and most often faster, than hand-written data structure code.

We have also evaluated our work. We noted decisions we made that did not turn out to be for the best. We discussed both conceptual and implementation flaws of our work. Where appropriate, we also discussed alternative approaches which might overcome the limitations we have observed.

8.2 Future work

This research has spawned many ideas for future work. A few of the more important are:

More layers. The number of additional layers (and implementations of layers) that could be written is unbounded. In our work we have uncovered no data structure that we feel could not be encapsulated as an NPT. Some of the more interesting layers we would like to investigate are hashing, concurrency, reference counting, assertions, and arbitrary graphs.

A second-generation tool. Our research group is currently constructing a second-generation Predator (P2) [Bat93a]. P2 is designed to improve on many of the defects discussed in Chapter 7. P2 is also better structured to allow for easier writing of new layers.

A formal model. Now that we have demonstrated the feasibility of NPTs in data structure programming, we feel it is important to classify our layers and their interactions via a more formal algebra. Such formalism would help us understand both the interactions and the meaning of semantic equivalence among layers.

Domain-specific optimizations. We feel that domain-specific optimizations are a key to further improving the efficiency of our generated code. In our research, we have only touched the surface of the optimizations we believe possible. We require a much better understanding of the general mechanisms needed to exploit each class of optimization.

Tool-based selection of type equations. Currently, the selection of the "proper" data structure for a given program is somewhat of a black art. Our approach allows for the substitution of implementations after a program is written. We envision a tool which would help programmers traverse the tree of possible implementations, and would assist in the selection of candidate type equations. A similar tool [Bat92c] has already been implemented for the Genesis system.

Further validation. We believe that our effort must be further validated. Of particular interest is more productivity experiments, the results of which would help us refine our **DS** and **LINK** interfaces to better serve the needs of application programmers. Also, the automated RL compiler must be completed to support every feature of the OPS5 language. The RL compiler is nearly complete and should be completed soon.

8.3 Contributions

In Chapter 1, we introduced six characteristics that we believe a system should possess to properly address the data structure programming problem. We feel our work with NPTs and Predator has made contributions in each of these areas:

Simplified programming. All of the sample Predator programming we (and our test subjects) wrote took less time than the equivalent programming written either by hand or with other component libraries. We attribute the increase in productivity to several factors. Chief among these is the consistent, high-level interface of the **DS** and **LINK** realms. Once a programmer learns and understands the interface for a given realm, they can easily program almost any task involving any layers of that realm. This is in contrast to more traditional component libraries for which programmers must be constantly learning new interfaces for new components.

Use data structures without knowing implementation details. This is a common benefit of component-based approaches, which our model shares. Our high-level interface and declarative type equations ensure that programmers can declare and use data structures without knowing how the layers are implemented. Several of our validation programs were written by programmers who did not understand how to implement the data structures they were using.

Debugging support. Predator optionally allows programmers to insert debugging statements in their generated output, which causes debugging to be performed on the original .dac

file. Thus, programmers never need to view the Predator-generated code which implements the high-level realm interface, and they view all primitive functions (such as **INSERT**, **DELETE** etc.) as atomic operations in the debugger. We have demonstrated this capability in three different debuggers (for three different operating systems), and do not envision any impediments towards implementing this support as a standard feature.

Changing the underlying implementation. Since all data structures (and links between data structures) are represented by declarative type equations, programmers may change the implementation of data structures simply and quickly. Our model further provides for a compiled environment, so the new, customized components required for the new type equations can be fabricated quickly by the applications programmer, rather than by a component author. Further, we have demonstrated the usefulness of "plug-and-play data structures" by comparing the time required to change the implementation of several of our validation experiments, and comparing that time with the time required for a traditional recoding effort. Also, we have shown that experimenting with multiple implementations in an existing program is desirable and easy with our model. We see this as a major contribution from our work.

Generation of efficient code. We believe that the combination of consistent realm interfaces, compiled environments, and domain-specific optimizations has helped us achieve acceptable efficiency in our generated code. As stated previously, without efficient code generation, our effort may be interesting, but is not practical. By expending considerable effort in the areas of code efficiency, we believe that our work stands as a foundation for eventual technology which can be used in commercial applications.

Evolution. The consistent realm interfaces we provide in our model allows our layer implementation code to be extremely regular. The type equation mechanism we use for specifying implementations forces all layer modules to export the same snippet function interface. Finally, we provide a specialization mechanism for augmenting base interfaces via our macro facility. Given all of these features, we have been able to add new layers and functions to existing Predator applications without having to rewrite either existing application code or existing layer implementation modules. We see our regular framework as a crucial base to allow NPT systems to be maintained and to evolve.

A major achievement of our work has been to lay the foundation which is making P++ [Sin93] possible. P++ is a set of extensions to the C++ language, which should enable programmers to construct GenVoca system generators much more easily. In a sense, P++ will be a

generator of “system generators.” P++ borrows heavily on the lessons learned from this research.

Another contribution of this work that we did not originally consider is that NPTs (and Predator) have helped us further refine the GenVoca domain modelling concepts. With each new GenVoca system constructed, we are able to further understand and refine our overall domain model.

We also feel that our work has made contributions in the understanding of the scalability of component libraries. We estimate that it would take approximately 40 NPT components to provide equivalent functionality of component libraries [Boo90, Lea88] which contain 400 components, or more. Further, the vertical parameterization of NPTs would allow those 40 components to be composed into additional structures which do not exist in those other libraries. We feel this is a major advantage, because our components are of comparable complexity with those in the other component libraries.

In summary, we defined an original set of goals for this research which were ambitious. We feel that our efforts have resulted in a base model and technology which shows a great deal of promise in further exploration into the generation of software systems for data structures.

Bibliography

[Abb92] M. Abbott, and L. Peterson. *A Language-Based Approach to Protocol Implementation*. Technical Report 92-2, University of Arizona, 1992.

[Acm91] ACM. Next generation database systems. *Communications of the ACM*, 34(10), October 1991.

[Aho83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass, 1983.

[Bal85] R. Balzer. A Fifteen-Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering*, SE 11, November, 1985.

[Bat88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, November 1988.

[Bat90] D. Batory. *The Genesis Database System Compiler: User Manual*. Technical Report 90-27, University of Texas, 1990.

[Bat92a] D. Batory, V. Singhal, and M. Sirkin. Implementing a domain model for data structures. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):375-402, September 1992.

[Bat92b] D. Batory, and S. O'Malley.The Design and Implementation of Hierarchical Software Systems Using Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bat92c] D. Batory, and J. Barnett. DaTE: The Genesis DBMS Software Layout Editor. In *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, P. Loucopoulos and R. Zicari, editors, John Wiley & Sons, New York, 1992.

[Bat93a] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable Software Architectures. In *Proceedings of the ACM SIGSOFT 93: Symposium on the Foundations of Software Engineering*, December 1993.

[Bat93b] D. Batory and D. Vasavada. Software Components for Object-Oriented Database Systems. *International Journal of Software Engineering and Knowledge Engineering* 3(2):165-192, 1993.

[Big89a] T. Biggerstaff, and A. Perlis. *Software Reusability. Vol I, Concepts and Models*. ACM Press, New York, 1989.

[Big89b] T. Biggerstaff, and A. Perlis. *Software Reusability. Vol II, Applications and Experience*. ACM Press, New York, 1989.

[Bla86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proceedings for the First ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 78-86, October 1986.

[Bla87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, January 1987.

[Boo87] G. Booch. *Software Components with Ada*. Benjamin/Cummings, Menlo Park, CA, 1987.

[Boo90] G. Booch, and M. Vilot. The Design of the C++ Booch Components. *In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications '90*, ACM Press, New York, 1990.

[Bor92] *OWL Classes*. Borland C++ User's Manual. Borland Inc, 1992.

[Bra93a] D. Brant. *Inferencing on Large Data Sets*. Ph.D. dissertation, Department of Computer Sciences, University of Texas, May 1993.

[Bra93b] D. Brant, and D. Miranker. Index support for rule activation. In *Proceedings of 1993 ACM SIGMOD*, May 1993.

[Cam92] R. Campbell, N. Islam, and P. Madany. Choices, frameworks and refinement. *Computing Systems*, 5(3):217-257, 1992.

[Cha89] C. Chambers, D. Unger, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications '89*, ACM Press, New York, 1989.

[Cog93] L. Coglianese, and R. Szymanski. DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD 1993*. Also Technical Report ADAGE-IBM-93-04, IBM Owego, May 1993.

[Coh89] D. Cohen. *AP5 Training Manual*. USC Information Sciences Institute, 1989.

[Coh90] S. Cohen. *Ada 9X Project Report*. 1990.

[Coh93] D. Cohen, and N. Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3):53-60, May 1993.

[Dat83] C. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass, 1983.

[Dav92] J. Davidson. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, February 1992.

[Fon90] M. Fontana, L. Oren, and M. Neath. *COOL — C++ object-oriented library*. Texas Instruments, 1990.

[For81] C. Forgy. *OPS5 User's Manual*. Technical Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[Gar92] D. Garlan, G. Kaiser, and D. Notkin. Composing Systems Using Tool Abstraction. *IEEE Computer*, June 1992.

[Gar93] D. Garlan, and M. Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering*, Vol 1, World Scientific Publishing Company, New York, 1993.

[Ghe87] C. Ghezzi, and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, New York, 1987.

[Gog86] J. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, 19(2):16-28, February 1986.

[Gol83] A. Goldberg, and D. Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley, Reading, Mass, 1983.

[Gor87] K. Gorlen. An Object-Oriented Class Library for C++ Programs. *USENIX C++ Conference*, 1987.

[Gor90] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, New York, 1990.

[Gri90] D. Gries, and D. Volpano. *The Transform - a New Language Construct*. Structured Programming, Springer-Verlag, New York, 1990.

[Haa90] L. Haas et al. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions of Knowledge and Data Engineering*, March 1990.

[Hab76] A. Habermann, L. Flon, and L. Cooprider. *Modularization and Hierarchy in a Family of Operating Systems*. *Communications of the ACM*, 19(5), May 1976.

[Har92] S. Harbison. *Modula-3*. Prentice-Hall, New York, 1992.

[Hei90] J. Heidemann, and G. Popek. *An extensible, stackable method of file system development*. Technical Report CSD-9000044, University of California, Los Angeles, December 1990.

[Hei91] J. Heidemann, and G. Popek. *A layered approach to file system development*. Technical Report CSD-910007, University of California, Los Angeles, March 1991.

[Hor84] E. Horowitz, and J. Munson. An Expansive View of Reusable Software. *IEEE Transactions on Software Engineering*, SE-10(5):477-487, September 1984.

[Hut91] N. Hutchinson, and L. Peterson. The *x*-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1), January 1991.

[Kai89] G. Kaiser, and D. Garlan. Synthesizing Programming Environments From Reusable Features. In *Software Reusability*, Vol II, Applications and Experience, ACM Press, New York, 1989.

[Kic92] G. Kiczales, and J. Lamping. Issues in the Design and Specification of Class Libraries. *In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications '92*, ACM Press, New York, 1992.

[Kor91] H. Korth, and A. Silberschatz. *Database System Concepts*. McGraw-Hill, New York, 1991.

[Kru92] C. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2), June 1992.

[Lam91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, October 1991.

[Lea88] D. Lea. libg++, The GNU C++ Library. *USENIX C++ Conference*, 1988.

[McD78a] J. McDermot, A. Newall, and J. Moore. The efficiency of Certain Production Systems. *Pattern Directed Inference Systems*, Waterman, Hayes, Roth (ed), Academic Press, New York, 1978.

[McD78b] J. McDermott, and C. Forgy. Production System Conflict Resolution Strategies. *Pattern Directed Inference Systems*, Waterman, Hayes, Roth (ed), Academic Press, New York, 1978.

[McN86a] D. McNicoll, C. Palmer, et al. *Common Ada Missile Packages (CAMP) Volume I: Overview and Commonality Study Results*. AFATL-TR-85-93, May 1986.

[McN86b] D. McNicoll, C. Palmer, et al. *Common Ada Missile Packages (CAMP) Volume II: Software Parts Composition Study Results*. AFATL-TR-85-93, May 1986.

[Mey88] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall, New York, 1989.

[Mir90] D. Miranker, D. Brant, B. Lofaso, and D. Gadbois. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, pages 685-692, 1990.

[Mir91] D. Miranker, and B. Lofaso. The Organization and Performance of a TREAT-Based Production System Compiler. *IEEE Transactions on Knowledge and Data Engineering*, pages 42-48, 1991.

[Nei80] J. Neighbors. *Software Construction Using Components*. Ph.D. dissertation, Technical Report 160, University of California, Irvine, 1980.

[Nei84] J. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, WE-10(5): 564-574, September 1984.

[Nov83] G. Novak. GLISP: A LISP-Based Programming System with Data Abstraction. AI Magazine, 4(3): 37-47, Fall 1983.

[Nov92] G. Novak. Software Reuse through View Type Clusters. In *Proceedings of the 7th Knowledge-Based Software Engineering Conference (KBSE-92)*, 1992.

[OMa90] S. O'Malley, and L. Peterson. *A new methodology for designing network software.* Technical Report 90-29, University of Arizona, September 1990.

[Pag90] T. Page, G. Popek, R. Guy, and J. Heidemann. *The Ficus Distributed File System: Replication via Stackable Layers*. Technical Report CSD-900009, University of California, Los Angeles, April 1990.

[Pal90] C. Palmer, and S. Cohen. Engineering and Applications of Reusable Software Resources. *Aerospace Software Engineering: A Collection of Concepts*, ed. C. Anderson and M. Dorfman. Vol. 136, *Progress in Astronautics and Aeronautics*, 1990.

[Par76] D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, March 1976.

[Par83] H. Partsch, and R. Steinbrüggen. Program Transformation Systems. In *ACM Computing Surveys*, 15(3):199-236, 1983.

[Pet90] L. Peterson, N. Hutchinson, H. Rao, and S. O'Malley. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer (Special Issue on Operating Systems)*, May 1990.

[Pop91] S. Popovich, S. Wu, and G. Kaiser. An Object-Based Approach to Implementing Distributed Concurrency Control. In *11th International Conference on Distributed Computing Systems*, pages 65-72, May 1991.

[Pri86] R. Prieto-Díaz, and J. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307-334, November 1986.

[Pri91] R. Prieto-Díaz, and G. Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

[Raj89] R. Raj, and H. Levy. *A Compositional Model for Software Reuse*. The Computer Journal. 32(4), August 1989.

[Ric79] C. Rich, H. Shrobe, and R. Waters. Overview of the Programmer's Apprentice. In *Proceedings of 6th International Joint Conference on Artificial Intelligence*, 1979.

[Ric90] C. Rich, and R. Waters. *The Programmer's Apprentice*. ACM Press, New York, 1990.

[Roy91] J. Roy. *Design and Use of the Jupiter File Management System*. M.Sc. Thesis, Department of Computer Sciences, University of Texas, 1991.

[Sch77] J. Schmidt. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems*, 1977.

[Sch79] E. Schonberg, J. Schwartz, and M. Sharir. Automatic Data Structure Selection in SETL. In *Proceedings of 6th ACM Symposium on Principles of Programming Languages*, pages 197-210, 1979.

[Sin93] V. Singhal, and D. Batory. P++: A Language for Software System Generators. Technical Report, Department of Computer Sciences, University of Texas at Austin, September 1993.

[Sir93] M. Sirkin, D. Batory, and V. Singhal. Software components in a data structure precompiler. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.

[Sno89] R. Snodgrass. *The Interface Description Language, 2nd edition*. Addison-Wesley, Reading, Mass, 1989.

[Str91] B. Stroustrup. *The C++ Programming Language, 2nd edition*. Addison-Wesley, Reading, Mass, 1991.

[Sul92] K. Sullivan, and D. Notkin. Reconciling Environment Integration and Software Evolution. *ACM TOSEM*, 1(3), July 1992.

[Tan89] D. Taenzer, M. Ganti, and S. Podar. Problems in Object-Oriented Software Reuse. *In Proceedings of the European Conference on Object-Oriented Programming '89*, 1989.

[Tra93] W. Tracz. LILEANNA: A Parameterized Programming Language. In *Proceedings of the 2nd International Workshop on Software Reuse*, March 1993.

[Vol85] D. Volpano, and R. Kieburtz. Software Templates. In *Proceedings of the 7th International Conference on Software Engineering*, 1985.

[Wei90] D. Weiss. *Synthesis Operational Scenarios*. Software Productivity Consortium, Inc. 1990.

Appendix A

Productivity experiments

This appendix contains the descriptions of the four productivity experiments:

A.1 Arrays

PURPOSE

The purpose of this experiment is to compare, in a very simple example, both the time it takes to write and the efficiency of code generated by the PREDATOR precompiler and code written by you, a sample programmer.

The program to be written has been kept purposefully simple. There is but one data structure - the array. The operations to be performed on the array are not complicated.

The basic concept of this experiment is that most programs are built from simple, smaller components. This small example should give me some insight into the methods that different programmers use to achieve the same result.

IMPORTANT NOTE: All programs will be kept completely anonymous. No names will ever be used without permission, and names will NEVER be tied to specific programming solutions.

PROGRAMMING METHOD

In programming this example, I am asking that you write, compile and test a solution to the problem presented below. You are asked to write your first solution without worrying about efficiency. Write your first solution such that it works. That is not to say that you should

not try to write it to be efficient. You should write it the way you normally would. Do not worry too much about finding faster ways to do it.

Please time yourself. Do not try to rush to finish the program, but rather keep an approximate time that you spent getting the program to work. I'm curious if it takes 10 minutes, 1/2 hour, 1 hour or six hours, etc...

Please save the source (.C) file for this program. I will need this for time comparisons.

After you have a working program, you may *optionally* try to improve the efficiency of the program. If you do, please keep track of the additional amount of time you spend on improving it, and please keep a copy of the source file you end up with.

When you are done, please send me a copy of the source file(s), and the time(s) it took to program them.

PROGRAM DESCRIPTION

The program is very simple. The source code provided in the next section should be used to set up the data structures. The EMPTYPE structure is a record containing information about an employee. The RECORD structure is an employee, augmented with a "delete flag". This is used to show if an employee record is deleted, or not.

The array, rawdata array is an array of employee records, used for input for the program.

Your program should do the following:

1. You should declare an array of RECORDS, large enough to hold all the data in the rawdata array.
2. You should insert all of the data from the rawdata array into your new array. Remember that you also need to initialize the delete flag to "0" for each record to show that the record is not deleted.
3. You should iterate through your new array. For each (of the 8) employees, if they are not deleted (via the delete flag) you should print their name with the following statement:

```
printf("%s\n", /* reference to name here */);
```

4. You should iterate through all of the employees again. For each customer whose age is over 40, you should print their name, and then delete them (with the delete flag).
5. You should make one final pass of the employees. Print the name of the employees who are not deleted.
6. **NOTE:** There is a printf header at the start of each print loop. See sample output for the text.

SAMPLE OUTPUT

The following is the output your program should produce:

```
All employees are:
Akers, Mark
Akin, Monica
Alexander, Joe
Anderson, Gwyn
Anderson, Mary
Anderson, Suzanne
Andrews, John
Andrews, Kay

Older employees are:
Akers, Mark
Akin, Monica
Anderson, Mary
Anderson, Suzanne
Andrews, John

Remaining employees are:
Alexander, Joe
Anderson, Gwyn
Andrews, Kay
```

A.2 Queues

PURPOSE

This experiment is a continuation of the PREDATOR/human programming tests. Once again you will be asked to write and debug a fairly simple program. I will then compare both the time it takes to write and the efficiency of the hand-written programs versus the same program written with PREDATOR.

In this experiment you will be asked to write a program involving queues. You will be provided with a simple abstract data type module for queues. **You are free to use this module, or not, as you see fit. In addition, you may also cannibalize the module, if you wish**.

The basic concept of this experiment is that off the shelf components are often used to help simplify the programming task. But often they do not provide the full functionality needed by the programmer. Thus, the programmer (you in this case) must decide how to proceed: use the module as is, ignore the module, or use parts of the module.

IMPORTANT NOTE: All programs will be kept completely anonymous. No names will ever be used without permission, and names will NEVER be tied to specific programming solutions.

PROGRAMMING METHOD

In programming this example, I am asking that you write, compile and test a solution to the problem presented below. Write a working solution to the problem. Before you begin you should decide if you want to write a program that is as efficient as possible, or write one that you can finish as quickly as possible. Both are acceptable solutions. Once you have decided which approach you want to take, please inform us (by E-mail) of your decision. If too many people pick one method we may have to request a few people to use their second choice solution method.

Please time yourself. Do not try to rush to finish the program, but rather, keep (an approximate) time that you spent getting the program to work. I'm curious if it takes 10 minutes, 1/2 hour, 1 hour or six hours, etc...

Please save the source (.C) file for this program. I will need this for time comparisons.

PROGRAM DESCRIPTION

This program is a supermarket simulation. You have a total of five active cash registers, each checking out customers. Each supermarket line is nothing more than a FIFO queue. A queue module has been written and is provided for your optional use. The program must perform several operations on the queues and the customers in them.

The data structures for this program are fairly simple. A customer ELEMENT is a structure containing all of the information that is interesting about the customer: their name, how

many items they have, and an array of (up to 50) items. Items are stored as integers in this simulation.

For those who decide to implement their own queues - you must make the queue so that it can grow as large as necessary - this is why it is implemented as a linked list in the provided module. For those who wish to use the QUEUE module as is - remember that all of the program can be performed with the operations provided.

A sample data file “exp2.in” is provided. Each line of this file is a customer that is in one of the checkout lines at the start of the program. An input routine is provided **“setup_simulation()”,** which will read the input data, and put it in the data structure for you. **NOTE**: If you write your own queue module you may have to modify this routine. Another routine **“write_output”** is also provided to write your data to an output file in a standard form. Again, you may have to modify this routine if you write your own queue module. The write_output routine creates an output file named “exp2.out”. You can use the provided output file “exp2_correct.out” for comparison. Your output file should march the provided file *exactly* when your solution is working properly.

Once you have read in the initial data, your program should perform the following steps in the order they are listed. At the end of the program you should write the output data out to the output file. The tasks your program must perform are:

1. Add customer Harold Hill with 10 items: 80 90 60 61 62 64 65 67 66 68 to lane 1.
2. Add customer Marion Librarian with 1 item: 13 to lane 1.
3. The first customer in lane 0 finishes checking out and leaves the store.
4. Customer Nancy Reagan (with 3 items: 8 8 8) cuts in line behind husband Ron (in front of George Bush).
5. Every 3rd customer in lane 1 decides to buy the promotional candy bar (item 234) and adds it to their cart. So, the first customer in line buys the item, the second and third don’t, the fourth does etc.
6. Customer George Bush in lane 3 realizes he has forgotten some items he needs, and leaves the shopping lane.
7. Lane 4, an 8 item or fewer lane, opens up. Each customer from lane 1who has 8 or fewer items moves over to lane 4. They do this in reverse order, so the customer nearest

the end of lane 1 who can move over does so. The customer nearest the front of lane 1 moves over last.

The QUEUE module

A sample queue module *(queue.h* and *q_funcs.c*) has been written for your use. The following functions are defined for in the queue module:

1. **void init_queue(QUEUE *the_queue);** - This function initializes the queue. It should be called once for each queue, before the queue is used.
2. **int enqueue(QUEUE *the_queue, ELEMENT *the_element);** - Adds an ELEMENT onto a QUEUE. Returns 1 (or TRUE) if the enqueue succeeds, or 0 otherwise.
3. **ELEMENT *dequeue(QUEUE *the_queue);** - Removes and returns a pointer to an item from the QUEUE. Returns NULL if the queue is empty.
4. **int queue_size(QUEUE *the_queue);** - Returns an integer representing the size of the queue.
5. **int is_empty(QUEUE *the_queue);** - Returns a boolean to inform if the queue is empty, or not.
6. **int is_full(QUEUE *the_queue);** - Returns a boolean to inform if the queue is full, or not.
7. **ELEMENT *make_elem(void);** - Creates a new element, and returns a pointer to it.
8. **void destroy_elem(ELEMENT **the_elem);** - Destroys an element. It is passed a the address of a pointer to an element.
9. **void setup_simulation(void)** - Initializes the data and reads in the initial values from an input file.
10. **void write_output(void)** - Writes the current simulation state to an output file in a standardized format.

SAMPLE OUTPUT

The following is the output your program should produce

```
Lane 0:
        1. Gore, Al. 1 item (999)
        2. Brown, Ron. 1 item (999)
```

```
Lane 1:
        1. Smith, John. 16 items (3 151 3 1 4 6 37 82 7 12 13 76 75
                                  74 73 234)
        2. Smith, Mary. 12 items (2 200 2 200 1 100 1 100 3 300 3
                                  300)
        3. Hill, Harold. 11 items (80 90 60 61 62 64 65 67 66 68
                                   234)

Lane 2:
        1. Doe, Jane. 10 items (121 5 5 5 5 5 5 5 9 10)
        2. Sluggo, Mr. 6 items (8 101 4 14 111 542)
        3. Bill, Mr. 3 items (16 22 28)

Lane 3:
        1. Reagan, Ron. 3 items (9 9 9)
        2. Reagan, Nancy. 3 items (8 8 8)
        3. Quayle, Dan. 2 items (222 333)

Lane 4:
        1. Librarian, Marion. 1 item (13)
        2. Andrews, Fred. 2 items (67 76)
```

A.3 Multiple lists

PURPOSE

This experiment is a continuation of the PREDATOR/human programming tests. Once again you will be asked to write and debug a fairly simple program. I will then compare both the time it takes to write and the efficiency of the hand-written programs versus the same program written with PREDATOR.

In this experiment you will be asked to write a program involving linked lists. You will be provided with a simple data structure for representing bank customers, and an array of initial data for your program. You will be asked to write some simple routines for ordered linked lists. The interesting twist is that you will have to maintain three distinct orderings on the same data. How you write your program to accomplish this is entirely up to you.

The idea behind this experiment is that many programming tasks involve repetitive operations on similar, but slightly different data structures. What I wish to observe is how different programmers approach this problem, and the relative coding speeds of the different approaches.

IMPORTANT NOTE: Once again, all programs will be kept completely anonymous. No names will ever be used without permission, and names will NEVER be tied to specific programming solutions.

PROGRAMMING METHOD

In programming this example, I am asking that you write, compile and test a solution to the problem presented below. Write a working solution to the problem. Please time yourself. Do not try to rush to finish the program, but rather, keep (an approximate) time that you spent getting the program to work. I'm curious if it takes 10 minutes, 1/2 hour, 1 hour or six hours, etc...

Please save the source (.C) file for this program. I will need this for time comparisons.

After you have a working program, you may *optionally* try to improve the efficiency of the program. If you do, please keep track of the additional amount of time you spend on improving it, and please keep a copy of the source file you end up with.

When you are done, please send me a copy of the source file(s), and the time(s) it took to program them.

PROGRAM DESCRIPTION

This program simply prints out reports on a number of bank customers. Your code will be responsible for taking the data from the static array provided, putting it in a list data structure of your design. Your program must keep the data ordered by name, account number and balance (three distinct orderings). After all the data is stored in your data structure you must perform the following operations:

1. Add a new customer: Nancy Reagan with account number 23 and a balance of: 6700000.
2. Change George Bush'es account number to: 56.
3. Remove Dan Quayle as a customer.
4. Print (to the output file exp3.out) the list of customers in alphabetic order.
5. Print the list in account order.
6. Print the list in balance order.

NOTE: The program template opens an output file with the handle “out_file”. You can write directly to that file handle.

NOTE: The following printf statement should be used to print each customer for each list. You will have to substitute your reference for the data structure fields.

```
fprintf(out_file, “Name: %s, %s  Account: %d  Balance: %d\n”,
<ref>.l_name, <ref>.f_name, <ref>.acct_num, <ref>.balance);
```

NOTE: You should place one blank line between each list in the output file.

You can use the provided output file “exp3_correct.out” for comparison. Your output file should march the provided file *exactly* when your solution is working properly.

SAMPLE OUTPUT

The following is the output your program should produce:

```
Name: Anderson, Tim  Account: 2  Balance: 5000
Name: Andrews, Joe  Account: 6  Balance: 4202020
Name: Andrews, Kay  Account: 5  Balance: 10000
Name: Bird, Larry  Account: 11  Balance: 32020
Name: Bush, George  Account: 56  Balance: 200023400
Name: Clinton, Bill  Account: 12  Balance: 30000001
Name: Gore, Al  Account: 7  Balance: 30000000
Name: Hill, Harold  Account: 1  Balance: 100
Name: Librarian, Maria  Account: 13  Balance: 89
Name: Librarian, Marion  Account: 10  Balance: 90
Name: Reagan, Nancy  Account: 23  Balance: 6700000
Name: Reagan, Ron  Account: 3  Balance: 400000000
Name: Young, Fred  Account: 4  Balance: 250000

Name: Hill, Harold  Account: 1  Balance: 100
Name: Anderson, Tim  Account: 2  Balance: 5000
Name: Reagan, Ron  Account: 3  Balance: 400000000
Name: Young, Fred  Account: 4  Balance: 250000
Name: Andrews, Kay  Account: 5  Balance: 10000
Name: Andrews, Joe  Account: 6  Balance: 4202020
Name: Gore, Al  Account: 7  Balance: 30000000
Name: Librarian, Marion  Account: 10  Balance: 90
Name: Bird, Larry  Account: 11  Balance: 32020
Name: Clinton, Bill  Account: 12  Balance: 30000001
Name: Librarian, Maria  Account: 13  Balance: 89
Name: Reagan, Nancy  Account: 23  Balance: 6700000
Name: Bush, George  Account: 56  Balance: 200023400

Name: Librarian, Maria  Account: 13  Balance: 89
Name: Librarian, Marion  Account: 10  Balance: 90
Name: Hill, Harold  Account: 1  Balance: 100
```

```
Name: Anderson, Tim  Account: 2  Balance: 5000
Name: Andrews, Kay  Account: 5  Balance: 10000
Name: Bird, Larry  Account: 11  Balance: 32020
Name: Young, Fred  Account: 4  Balance: 250000
Name: Andrews, Joe  Account: 6  Balance: 4202020
Name: Reagan, Nancy  Account: 23  Balance: 6700000
Name: Gore, Al  Account: 7  Balance: 30000000
Name: Clinton, Bill  Account: 12  Balance: 30000001
Name: Bush, George  Account: 56  Balance: 200023400
Name: Reagan, Ron  Account: 3  Balance: 400000000
```

A.4 Interlinked data structures

PURPOSE

This experiment is a continuation of the PREDATOR/human programming tests. Once again you will be asked to write and debug a fairly simple program. I will then compare both the time it takes to write and the efficiency of the hand-written programs versus the same program written with PREDATOR.

In this experiment you will be asked to write a program involving binary trees, lists, and links. You will be asked to write an efficient implementation of multiple data structures which are interconnected.

IMPORTANT NOTE: Once again, all programs will be kept completely anonymous. No names will ever be used without permission, and names will NEVER be tied to specific programming solutions.

PROGRAMMING METHOD

In programming this example, I am asking that you write, compile and test a solution to the problem presented below. Write a working solution to the problem. Please time yourself. Do not try to rush to finish the program, but rather, keep (an approximate) time that you spent getting the program to work. I'm curious if it takes 10 minutes, 1/2 hour, 1 hour or six hours, etc...

Please save the source (.C) file for this program. I will need this for time comparisons.

After you have a working program, you may *optionally* try to improve the efficiency of the program. If you do, please keep track of the additional amount of time you spend on improving it, and please keep a copy of the source file you end up with.

When you are done, please send me a copy of the source file(s), and the time(s) it took to program them.

PROGRAM DESCRIPTION

This program involves binary trees, linked lists and links.You will be provided with two simple data structures - one for company employees and one for departments within the company. You will also be provided with initial data to load into these data structures. Your programming task will be to add, delete and traverse these data structures, while using the ordering of the list and binary tree, and using links.

Links require some explanation. A *link* is a connection between records in two different data structures based on a *predicate* involving fields of the two records.In this assignment you will maintain two linkages between the employee and department data structures.

1. The *works_in* link connects each employee record to the department record such that the department number of the employee equals the department number of the department record.

2. The *manages* link connects departments to one employee record such that the manager field of the department record equals the emp_id field of the employee record.

Note that links are the same as database joins. For this assignment you must write your code so that the links are maintained by insertions and deletions and updates. It is possible to implement links by performing a cross-product on the two data structures, but for this assignment you must implement links using pointers in the data structures themselves.

In this assignment you should implement the department data structure with a linked list, ordered by department number. The employees should be organized in a binary tree, ordered by name (last and then first).

Your program should begin by reading the input data into the two data structures, and building the two links between them. Then your program should do the following:

1. Print out all the employees (their names and employee IDs) in departments 1 and 3. Use the following printf statements for each department and then each employee.

```
fprintf(out_file, "Dept name: %s\n", <ref>.dept_name);

fprintf(out_file, "   Name: %s, %s  Emp num: %d\n", <ref>.l_name,
        <ref>.f_name, <ref>.emp_id);
```

2. Add another employee: Nancy Reagan, age 75, in department 4 with employee id 20.
3. Maria Librarian moves to department 2, and Bruce Babbit is promoted to department manager of department 6.
4. Print a list of department managers using the following fprintf statement.

```
fprintf(out_file, "Dept: %s  Manager: %s, %s\n", <ref>.dept_name,
        <ref>.l_name, <ref>.f_name);
```

5. Marion Librarian retires from the company, and is removed from the database.
6. Print a list of employees, by department. Use the statement in step 1 above to print the data.

NOTE: The program template opens an output file with the handle "out_file". You can write directly to that file handle.

NOTE: You should place one blank line between each list in the output file.

You can use the provided output file "exp4_correct.out" for comparison. Your output file should march the provided file *exactly* when your solution is working properly.

SAMPLE OUTPUT

The following is the output your program should produce:

```
Dept name: Accounting
 Name: Quayle, Dan Emp num: 9
 Name: Brown, Ron Emp num: 17
 Name: Andrews, Joe Emp num: 16

Dept name: Development
 Name: Bentsen, Lloyd Emp num: 3
 Name: Andrews, Kay Emp num: 5

Dept: Accounting Manager: Quayle, Dan
Dept: Receiving Manager: Cisneros, Henry
Dept: Development Manager: Bentsen, Lloyd
Dept: Shipping Manager: Clinton, Bill
Dept: Marketing Manager: Hill, Harold
Dept: Research Manager: Babbit, Bruce

Dept name: Accounting
```

```
 Name: Quayle, Dan Emp num: 9
 Name: Brown, Ron Emp num: 17
 Name: Andrews, Joe Emp num: 16
Dept name: Receiving
 Name: Librarian, Maria Emp num: 13
 Name: Cisneros, Henry Emp num: 6
 Name: Anderson, Tim Emp num: 2
Dept name: Development
 Name: Bentsen, Lloyd Emp num: 3
 Name: Andrews, Kay Emp num: 5
Dept name: Shipping
 Name: Reagan, Nancy Emp num: 20
 Name: Young, Fred Emp num: 4
 Name: Reagan, Ron Emp num: 14
 Name: Clinton, Bill Emp num: 12
Dept name: Marketing
 Name: Hill, Harold Emp num: 1
 Name: Gore, Al Emp num: 7
 Name: Bush, George Emp num: 8
Dept name: Research
 Name: Babbit, Bruce Emp num: 11
 Name: Bird, Larry Emp num: 15
```

Appendix B

LEAPS test cases

This appendix lists the OPS5 source code for the LEAPS test cases evaluated in this research, as well as a sample RL driver from the puz.ops test case:

B.1 LEAPS test cases

- The following is the source code for the basic_cycle.ops test case:

```
(literalize foo
 bar)

(p basic-cycle-timing-test
 (foo ^bar {<x> < 100000})
-->
 (modify 1 ^bar (compute <x> + 1))
)
```

- The following is the source code for the no_join1.ops test case:

```
(literalize foo
 bar )

(p no-join-test
 (foo ^bar {<x> < 500})
-->
 (write “bar has value” <x>)
 (modify 1 ^bar (compute <x> + 1 ))
)
```

- The following is the source code for the no_join2.ops test case:

```
(literalize foo
 bar )

(p extra-test
```

```
 (foo ^bar { < 5 > 2})
-->
 (write “bar has value between 2 and 5”)
)

(p no-join-test
 (foo ^bar {<x> < 10})
-->
 (write “bar has value” <x>)
 (modify 1 ^bar (compute <x> + 1 ))
)
```

- The following is the source code for the big_join.ops test case:

```
(literalize foo
 bar )

(literalize counter
 result )

(literalize rel1
 val )

(literalize rel2
 val )

(literalize marker foo)

(p generate-data
 (foo ^bar { <x> < 1000 })
-->
 (make rel1 ^val <x> )
 (make rel2 ^val <x> )
 (modify 1 ^bar (compute <x> + 1 ))
)

(p do-join
 (rel1 ^val <x>)
 (rel2 ^val <x>)
-->
 (make marker ^foo 0)
)

(p count-result
 (marker ^foo 0)
 (counter ^result <x>)
-->
 (modify 2 ^result (compute <x> + 1))
 (remove 1)
)
```

```
(p pr-result
 (counter ^result <y>)
-->
 (write <y> “tuples in the join”)
)
```

- The following is the source code for the puz.ops test case:

```
(literalize pile
 id )

(literalize piece
 pid
 color )

(literalize edge
 pid
 eid
 shape
 matched
 location )

(p put-pieces-in-pile
 (pile ^id <c>)
 (piece ^pid <pid> ^color <c>)
 (edge ^pid <pid> ^location start)
-->
 (modify 3 ^location <c>)
)

(p match-colors
 (pile ^id {<c> <> start})
 {<e1>(edge ^eid <id1> ^location <c> ^shape <s> ^matched false)}
 {<e2>(edge ^eid { <id2> <> <id1>} ^location <c> ^shape <s>
           ^matched false)}
-->
 (write “match edges” <id1> <id2>)
 (modify <e1> ^matched true)
 (modify <e2> ^matched true)
)

; once we have moved all the pieces out of the start pile and
; can’t match anymore within the color pile
; move the edges to the brute-force pile
; count on the resolution strategy to work breadth first
(p colors-done
 (pile ^id {<c> <> start})
 (edge ^eid <id> ^location <c> ^shape <s> ^matched false)
-->
 (modify 2 ^location brute-force)
)
```

```
; match whatever edges remain
(p do-brute-force
 (edge ^eid <id1> ^location brute-force ^shape <s> ^matched false)
 (edge ^eid {<id2> <> <id1>} ^location brute-force ^shape <s>
                  ^matched false)
-->
 (write “match edges” <id1> <id2>)
 (modify 1 ^matched true)
 (modify 2 ^matched true)
)
```

- The following is the source code for the triples.ops test case:

```
(literalize foo
 bar int)

(literalize rel1
 val int)

(literalize rel2
 val int)

(p generate-data
 (foo ^bar {<x> < 35})
-->
 (make rel1 ^val <x>)
 (make rel2 ^val <x>)
 (modify 1 ^bar (compute <x> + 1 ))
)

(p print-sequences
 (rel1 ^val <r>)
 (rel2 ^val {<s> > <r>})
 (rel1 ^val {<t> > <r> > <s>})
-->
 (write <r> <s> <t> “ is a strictly increasing sequence”)
)
```

- The following is the source code for the big_num.ops test case:

```
(literalize count val)

(literalize a b)

(p r1
 (count ^val {<x> < 10000})
-->
 (make a ^b <x>)
 (modify 1 ^val (compute <x> + 1))
)
```

```
(p r2
 (a ^b <x>)
 -(a ^b > <x>)
-->
 (write <x> “is the biggest number”)
)
```

- The following is the source code for the jig25.ops test case:

```
(literalize ppiece num edge match color side piled)

(p heap ;make pile of each color
 (*pile)
 (ppiece ^num <x> ^color <y> ^piled no)
 -(ppiece ^num <x> ^piled yes)
-->
 (call pile <x> <y>)
 (modify 2 ^piled yes)
)

(p heaped ;done pileing
 (*pile)
-->
 (remove 1)
 (make *cornr))

(p corner
 (*cornr)
 (ppiece ^num <x> ^edge * ^match no ^side <y>)
 (ppiece ^num <x> ^edge * ^match no ^side <> <y>)
-->
 (remove 1)
 (call corn <x>)
 (modify 2 ^match yes)
 (modify 3 ^match yes)
 (make *cb)
)

(p cb ; attach border pieces to corner
 (*cb)
 (ppiece ^num <x> ^edge * ^match yes ^side <a>)
 (ppiece ^num <x> ^edge * ^match yes ^side <> <a>)
 (ppiece ^num <x> ^edge {<y> <> *} ^match no)
 (ppiece ^num {<z> <> <x>} ^edge <y> ^match no)
 (ppiece ^num <z> ^edge * ^match no)
-->
 (call assemble <z> <x> <y>)
 (modify 4 ^match yes)
 (modify 5 ^match yes)
 (modify 6 ^match yes)
```

```
)

(p endcb ;done with corner
 (*cb)
 (ppiece ^num <x> ^edge * ^match yes)
 (ppiece ^num <x> ^edge * ^match yes)
 -(ppiece ^num <x> ^edge <> * ^match no)
-->
 (remove 1)
 (make *flg)
)

(p bordercorner ;assembles the border
 (*flg) ;working on border
 (ppiece ^num <x> ^edge * ^match yes)
 (ppiece ^num <x> ^edge {<z> <> *} ^match no)
 (ppiece ^num {<y> <> <x>} ^edge * ^match no ^side <a>)
 ^num <y> ^edge * ^match no ^side <> <a>)
 (ppiece ^num <y> ^edge <z> ^match no)
-->
 (call assemble <y> <x> <z>)
 (modify 3 ^match yes)
 (modify 4 ^match yes)
 (modify 6 ^match yes)
)

(p border ;assembles the border
 (*flg) ;working on border
 (ppiece ^num <x> ^edge * ^match yes)
 (ppiece ^num <x> ^edge {<z> <> *} ^match no)
 (ppiece ^num {<y> <> <x>} ^edge * ^match no ^side <a>)
 -(ppiece ^num <y> ^edge * match no ^side <> <a>)
 (ppiece ^num <y> ^edge <z> ^match no)
-->
 (call assemble <y> <x> <z>)
 (modify 3 ^match yes)
 (modify 4 ^match yes)
 (modify 5 ^match yes)
)

(p restofborder ;assembles the border
 (*flg) ;working on border
 (ppiece ^num <x> ^edge * ^match yes)
 (ppiece ^num <x> ^edge {<z> <> *} ^match no)
 (ppiece ^num {<y> <> <x>} ^edge * ^match yes)
 (ppiece ^num <y> ^edge <z> ^match no)
-->
 (call assemble <y> <x> <z>)
 (modify 3 ^match yes)
 (modify 5 ^match yes)
)
```

```
(p setcolor ;assemble by color
 (*flg) ;find flag
 -(ppiece ^edge << a i p w x y v o h f d b >> ^match no)
-->
 (remove 1)
 (make colorflg)
)

(p colors ;assembles by color
 (colorflg) ;find flag
 (ppiece ^num <x> ^color <z> ^match yes)
 (ppiece ^num {<y> <> <x>} ^color <z>)
 (ppiece ^num <x> ^edge {<w> <> *} ^match no)
 (ppiece ^num <y> ^edge <w> ^match no)
-->
 (call assemble <y> <x> <w>)
 (modify 4 ^match yes)
 (modify 5 ^match yes)
)

(p setins ;remove color flag
 (colorflg) ;find flag
 (ppiece ^num <x> ^edge <> * ^match no ^color <y>)
 -(ppiece ^num <> <x> ^edge <> * ^match no ^color <y>)
-->
 (remove 1)
 (make insflag)
)

(p insides ;assemble by exhaustive trial
 (insflag) ;find flag
 (ppiece ^num <y> ^match yes)
 (ppiece ^num <y> ^edge {<x> <> *} ^match no)
 (ppiece ^num {<z> <> <y>} ^edge <x> ^match no)
 -(ppiece ^num <z> ^match yes)
-->
 (call assemble <z> <y> <x>)
 (modify 3 ^match yes)
 (modify 4 ^match yes)
)
```

- The following is the source code for the waltz.ops test case:

```
(literalize stage value)
(literalize line p1 p2)
(literalize edge p1 p2 joined label plotted)
(literalize junction p1 p2 p3 base_point type)

(p begin
 (stage ^value start)
```

```
-->
 (write clr)
 (make line ^p1 0122 ^p2 0107)
 (make line ^p1 0107 ^p2 2207)
 (make line ^p1 2207 ^p2 3204)
 (make line ^p1 3204 ^p2 6404)
 (make line ^p1 2216 ^p2 2207)
 (make line ^p1 3213 ^p2 3204)
 (make line ^p1 2216 ^p2 3213)
 (make line ^p1 0107 ^p2 2601)
 (make line ^p1 2601 ^p2 7401)
 (make line ^p1 6404 ^p2 7401)
 (make line ^p1 3213 ^p2 6413)
 (make line ^p1 6413 ^p2 6404)
 (make line ^p1 7416 ^p2 7401)
 (make line ^p1 5216 ^p2 6413)
 (make line ^p1 2216 ^p2 5216)
 (make line ^p1 0122 ^p2 5222)
 (make line ^p1 5222 ^p2 7416)
 (make line ^p1 5222 ^p2 5216)
 (modify 1 ^value duplicate)
)

(p reverse_edges
 (stage ^value duplicate)
 (line ^p1 <p1> ^p2 <p2>)
-->
 (write draw <p1> <p2> (crlf))
 (make edge ^p1 <p1> ^p2 <p2> ^joined false)
 (make edge ^p1 <p2> ^p2 <p1> ^joined false)
 (write adding edge: <p1> <p2> and: <p2> <p1> (crlf))
 (remove 2)
)

(p done_reversing
 (stage ^value duplicate)
 -(line)
-->
 (modify 1 ^value detect_junctions)
)

(p make-3_junction
 (stage ^value detect_junctions)
 (edge ^p1 <base_point> ^p2 <p1> ^joined false)
 (edge ^p1 <base_point> ^p2 {<p2> <> <p1>} ^joined false)
 (edge ^p1 <base_point> ^p2 {<p3> <> <p1> <> <p2>} ^joined false)
-->
 (make junction
   ^type (make_3_junction <base_point> <p1> <p2> <p3>)
 ^base_point <base_point>)
```

```
 (write making a junction: <type> points: <base_point> <p1> <p2>
        <p3> (crlf))
 (modify 2 ^joined true)
 (modify 3 ^joined true)
 (modify 4 ^joined true)
)

(p make_L
 (stage ^value detect_junctions)
 (edge ^p1 <base_point> ^p2 <p2> ^joined false)
 (edge ^p1 <base_point> ^p2 {<p3> <> <p2>} ^joined false)
 -(edge ^p1 <base_point> ^p2 {<> <p2> <> <p3>})
-->
 (make junction
       ^type L
       ^base_point <base_point>
       ^p1 <p2>
       ^p2 <p3>)
 (write making a junction: L points: <base_point> <p1> <p2>
                                     (crlf))
 (modify 2 ^joined true)
 (modify 3 ^joined true)
)

(p done_detecting
 (stage ^value detect_junctions)
 -(edge ^joined false)
-->
 (modify 1 ^value find_initial_boundary)
)

(p initial_boundary_junction_L
 (stage ^value find_initial_boundary)
 (junction ^type L ^base_point <base_point> ^p1 <p1> ^p2 <p2>)
 (edge ^p1 <base_point> ^p2 <p1>)
 (edge ^p1 <base_point> ^p2 <p2>)
 -(junction ^base_point > <base_point>)
-->
 (write initial_boundary_junction_L <base_point> <p1> <p2> (crlf))
 (modify 3 ^label B)
 (modify 4 ^label B)
 (modify 1 ^value find_second_boundary)
)

(p initial_boundary_junction_arrow
 (stage ^value find_initial_boundary)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1>)
 (edge ^p1 <bp> ^p2 <p2>)
 (edge ^p1 <bp> ^p2 <p3>)
 -(junction ^base_point > <bp>)
```

```
-->
 (modify 3 ^label B)
 (modify 4 ^label +)
 (modify 5 ^label B)
 (modify 1 ^value find_second_boundary)
)

(p second_boundary_junction_L
 (stage ^value find_second_boundary)
 (junction ^type L ^base_point <base_point> ^p1 <p1> ^p2 <p2>)
 (edge ^p1 <base_point> ^p2 <p1>)
 (edge ^p1 <base_point> ^p2 <p2>)
 -(junction ^base_point < <base_point>)
-->
 (modify 3 ^label B)
 (modify 4 ^label B)
 (modify 1 ^value labeling)
)

(p second_boundary_junction_arrow
 (stage ^value find_second_boundary)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1>)
 (edge ^p1 <bp> ^p2 <p2>)
 (edge ^p1 <bp> ^p2 <p3>)
 -(junction ^base_point < <bp>)
-->
 (write second_boundary_junction_arrow <bp> <p1> <p2> <p3> (crlf))
 (modify 3 ^label B)
 (modify 4 ^label +)
 (modify 5 ^label B)
 (modify 1 ^value labeling)
)

(p match_edge
 (stage ^value labeling)
 (edge ^p1 <p1> ^p2 <p2> ^label {<label> << + - B >>})
 (edge ^p1 <p2> ^p2 <p1> ^label nil)
-->
 (modify 2 ^plotted t)
 (modify 3 ^label <label> ^plotted t)
 (write plot <label> <p1> <p2> (crlf))
)

(p label_L
 (stage ^value labeling)
 (junction ^type L ^base_point <p1>)
 (edge ^p1 <p1> ^p2 <p2> ^label << + - >>)
 (edge ^p1 <p1> ^p2 <> <p2> ^label nil)
-->
 (modify 4 ^label B)
```

```
)

(p label_tee_A
 (stage ^value labeling)
 (junction ^type tee ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1> ^label nil)
 (edge ^p1 <bp> ^p2 <p3>)
-->
 (modify 3 ^label B)
 (modify 4 ^label B)
)

(p label_tee_B
 (stage ^value labeling)
 (junction ^type tee ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1>)
 (edge ^p1 <bp> ^p2 <p3> ^label nil)
-->
 (modify 3 ^label B)
 (modify 4 ^label B)
)

(p label_fork-1
 (stage ^value labeling)
 (junction ^type fork ^base_point <bp>)
 (edge ^p1 <bp> ^p2 <p1> ^label +)
 (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label nil)
 (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>})
-->
 (modify 4 ^label +)
 (modify 5 ^label +)
)

(p label_fork-2
 (stage ^value labeling)
 (junction ^type fork ^base_point <bp>)
 (edge ^p1 <bp> ^p2 <p1> ^label B)
 (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label -)
 (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
-->
 (modify 5 ^label B)
)

(p label_fork-3
 (stage ^value labeling)
 (junction ^type fork ^base_point <bp>)
 (edge ^p1 <bp> ^p2 <p1> ^label B)
 (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label B)
 (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
-->
 (modify 5 ^label -)
```

```
)

(p label_fork-4
 (stage ^value labeling)
 (junction ^type fork ^base_point <bp>)
 (edge ^p1 <bp> ^p2 <p1> ^label -)
 (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label -)
 (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
-->
 (modify 5 ^label -)
)

(p label_arrow-1A
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1> ^label {<label> << B - >>})
 (edge ^p1 <bp> ^p2 <p2> ^label nil)
 (edge ^p1 <bp> ^p2 <p3>)
-->
 (modify 4 ^label +)
 (modify 5 ^label <label>)
)

(p label_arrow-1B
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1> ^label {<label> << B - >>})
 (edge ^p1 <bp> ^p2 <p2>)
 (edge ^p1 <bp> ^p2 <p3> ^label nil)
-->
 (modify 4 ^label +)
 (modify 5 ^label <label>)
)

(p label_arrow-2A
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p3> ^label {<label> << B - >>})
 (edge ^p1 <bp> ^p2 <p2> ^label nil)
 (edge ^p1 <bp> ^p2 <p1>)
-->
 (modify 4 ^label +)
 (modify 5 ^label <label>)
)

(p label_arrow-2B
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p3> ^label {<label> << B - >>})
 (edge ^p1 <bp> ^p2 <p2>)
 (edge ^p1 <bp> ^p2 <p1> ^label nil)
```

```
-->
 (modify 4 ^label +)
 (modify 5 ^label <label>)
)

(p label_arrow-3A
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1> ^label +)
 (edge ^p1 <bp> ^p2 <p2> ^label nil)
 (edge ^p1 <bp> ^p2 <p3>)
-->
 (modify 4 ^label -)
 (modify 5 ^label +)
)

(p label_arrow-3B
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p1> ^label +)
 (edge ^p1 <bp> ^p2 <p2>)
 (edge ^p1 <bp> ^p2 <p3> ^label nil)
-->
 (modify 4 ^label -)
 (modify 5 ^label +)
)

(p label_arrow-4A
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p3> ^label +)
 (edge ^p1 <bp> ^p2 <p2> ^label nil)
 (edge ^p1 <bp> ^p2 <p1>)
-->
 (modify 4 ^label -)
 (modify 5 ^label +)
)

(p label_arrow-4B
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p3> ^label +)
 (edge ^p1 <bp> ^p2 <p2>)
 (edge ^p1 <bp> ^p2 <p1> ^label nil)
-->
 (modify 4 ^label -)
 (modify 5 ^label +)
)

(p label_arrow-5A
 (stage ^value labeling)
```

```
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p2> ^label -)
 (edge ^p1 <bp> ^p2 <p1>)
 (edge ^p1 <bp> ^p2 <p3> ^label nil)
-->
 (modify 4 ^label +)
 (modify 5 ^label +)
)

(p label_arrow-5B
 (stage ^value labeling)
 (junction ^type arrow ^base_point <bp> ^p1 <p1> ^p2 <p2> ^p3 <p3>)
 (edge ^p1 <bp> ^p2 <p2> ^label -)
 (edge ^p1 <bp> ^p2 <p1> ^label nil)
 (edge ^p1 <bp> ^p2 <p3>)
-->
 (modify 4 ^label +)
 (modify 5 ^label +)
)

(p done_labeling
 (stage ^value labeling)
-->
 (modify 1 ^value plot_remaining_edges)
)

(p plot_remaining
 (stage ^value plot_remaining_edges)
 (edge ^plotted nil ^label {<label> <> nil} ^p1 <p1> ^p2 <p2>)
-->
 (write plot <label> <p1> <p2> (crlf))
 (modify 2 ^plotted t)
)

(p plot_boundaries
 (stage ^value plot_remaining_edges)
 (edge ^plotted nil ^label nil ^p1 <p1> ^p2 <p2>)
-->
 (write plot B <p1> <p2> (crlf))
 (modify 2 ^plotted t)
)

(p done_plotting
 (stage ^value plot_remaining_edges)
 -(edge ^plotted nil)
-->
 (modify 1 ^value done)
)

(p done
 (stage ^value done)
```

```
-->
 (write see trace.waltz for description of execution- hit CR to end
 (crlf))
 (halt)
)
```

B.2 RL driver for puz.ops test case

```
/*****************************************************************/
/*** Files to include.                                         ***/
/*****************************************************************/

#include <stdio.h>        /* Standard input and output library. */
#include <string.h>       /* The parser needs string functions. */

/*****************************************************************/
/*** Constants and variables for the run-time evaluator.       ***/
/*****************************************************************/

#define NEG_TIME -1          /* Timestamp before system starts. */
#define TRUE 1               /* Logical true. */
#define FALSE 0              /* Logical false. */

#define L_PAREN '('          /* Some useful characters. */
#define R_PAREN ')'
#define SPACE ' '
#define NULL_CHAR '\0'

#define RUN "run"            /* Useful tokens we may parese. */
#define MAKE "make"
#define EXIT "exit"

#define LINE_LEN 255         /* Input line length maximum. */
#define TOKEN_LEN 20         /* Length of longest input token. */

/*****************************************************************/
/*** Constants and variables for the production system.        ***/
/*****************************************************************/

#define NUM_RULES 4          /* How many rules in this system. */
#define TUPLE_SIZE 3         /* Size of the biggest join. */

#define START 2
#define A 3
#define B 4
#define C 5
#define D 6
```

```
#define E 7
#define F 8
#define G 9
#define H 10
#define I 11
#define J 12
#define K 13
#define L 14
#define BLUE 15
#define GREEN 16
#define WHITE 17
#define ORANGE 18
#define BRUTE_FORCE 19

                              /* Element definitions for system. */
struct pile
{
 int id;
};
struct piece
{
 int pid;
 int color;
};
struct edge
{
 int pid;
 int eid;
 int shape;
 int matched;
 int location;
};

                                /* Schema and cursor definitions */
SCHEMA pile_struct on element pile = predind[dlist[timest[
       persist[puz.per, 20000], GLOBAL]], id != START];
pile_struct cont0;
CURSOR c0 ON cont0 WHERE id != START;

SCHEMA piece_struct on element piece = dlist[timest[
       persist[puz.per, 20000], GLOBAL]];
piece_struct cont1;
CURSOR c1 ON cont1;

SCHEMA edge_struct on element edge = predind[dlist[timest[
       persist[puz.per, 20000], GLOBAL]], location == START];
edge_struct cont2;
CURSOR c2 ON cont2 WHERE location == START;
CURSOR c3 ON cont2;
```

```
                              /* Links and link cursors needed. */
LINK lnk1 ON MANY cont0 TO MANY cont1 USING NESTEDLOOP
     where cont0.id == cont1.color;
LCURSOR lc1 ON lnk1 USING c0, c1;
LINK lnk2 ON MANY cont1 TO MANY cont2 USING NESTEDLOOP
     where cont1.pid == cont2.pid;
LCURSOR lc2 ON lnk2 USING c1, c2;
LINK lnk3 ON MANY cont0 TO MANY cont2 USING NESTEDLOOP
     where cont0.id == cont2.location;
LCURSOR lc3 ON lnk3 USING c0, c2;
LCURSOR lc4 ON lnk3 USING c0, c3;

/*************************************************************/
/*** Global variables.                                     ***/
/*************************************************************/

int rule_to_fire;            /* Which is the next rule to fire? */
int dom_container;           /* Which container does the dom    */
                             /* object come from? */
int last_rule_fired;         /* What was the number of the last */
                             /* rule fired? */
int from_stack;              /* Is the dom object from the stack */
                             /* or the wait list? */

char inp_line[LINE_LEN];     /* Input line for parser. */
char token[TOKEN_LEN];       /* Input token for parser. */
char *t_ptr, *t_ptr2;        /* Parsing character pointers. */
int done;                    /* Are we done with the program? */

                             /* Structure for the wait stack.  */
                             /* The positions of the saved curs*/
                             /* The time stamp of most dom obj */
                             /* The cont of the most dom obj.  */
struct wss
{
 CURS_POS pos_arr[TUPLE_SIZE];
 int time_stamp;
 int cont;
};

                            /* Wait stack is just a linked list */
                            /* Only one wait stack declared. */
                            /* Cursor to wait stack declared. */
SCHEMA ws_struct on element wss = dlist[malloc[]];
ws_struct ws;
CURSOR wsc ON ws;


                              /* Structure for the wait stack. */
                              /* The position of the object. */
                              /* The container it comes from. */
```

```
                                /* Its timestamp. */
struct wls
{
 CURS_POS cursor_position;
 int cont;
 int time_stamp;
};

                               /* Wait list is just a linked list */
                               /* Only one wait list declared. */
                               /* Cursor to wait list declared. */
SCHEMA wl_struct on element wls = dlist[malloc[]];
wl_struct wl;
CURSOR wlc ON wl;

/*****************************************************************/
/*** Push/pop entries on the wait list or stack.              ***/
/*****************************************************************/

void push_wl(int cont, CURS_POS curs, int time_stamp)
{
 struct wls t_wl;              /* New wait list object. */

                               /* Fill the fields. */
                               /* Insert the wait list object. */
 t_wl.cont = cont;
 t_wl.cursor_position = curs;
 t_wl.time_stamp = time_stamp;
 INS(wl, t_wl, wlc);
}

void pop_wl(void)
{
                               /* Go to first object. */
                               /* If there is one... */
                               /* Delete it. */
                               /* Set up to next object. */
 RESET_CURSOR(wlc, CONT_START);
 if (LAST_CURS_OP(wlc) != EOR)
 {
    DEL(wlc);
 }
}

void push_ws(CURS_POS cp1, CURS_POS cp2, CURS_POS cp3,
          int time_stamp, int cont)
{
 struct wss t_ws;              /* New wait stack object. */

                               /* Fill the fields. */
                               /* Insert the wait stack object. */
```

```
 t_ws.pos_arr[0] = cp1;
 t_ws.pos_arr[1] = cp2;
 t_ws.pos_arr[2] = cp3;
 t_ws.time_stamp = time_stamp;
 t_ws.cont = cont;
 INS(ws, t_ws, wsc);
}

void pop_ws(void)
{
                              /* Go to first object. */
                              /* If there is one... */
                              /* Delete it. */
                              /* Set up to next object. */
 RESET_CURSOR(wsc, CONT_START);
 if (LAST_CURS_OP(wsc) != EOR)
 {
    DEL(wsc);
 }
}

/*****************************************************************/
/*** Determine container of most dominant object. ***/
/*****************************************************************/

int get_next_dom_object()
{
 int list_time, stack_time; /* Timestamp for wait list and wait */
                            /* stack objects. */

                            /* Assume no list or stack object. */
                            /* Go to most recent list and stack */
                            /* objects. */
 list_time = stack_time = NEG_TIME;
 RESET_CURSOR(wlc, CONT_START);
 RESET_CURSOR(wsc, CONT_START);

                            /* Set timestamp of list and stack */
                            /* objects (if they exist). */
 if (LAST_CURS_OP(wlc) != EOR)
    list_time = wlc.time_stamp;
 if (LAST_CURS_OP(wsc) != EOR)
    stack_time = wsc.time_stamp;

                            /* If stack object is most recent. */
 if (stack_time > list_time)
 {
                            /* Set the cursor positions. */
                            /* Set the stack flag. */
                            /* Return the dom object container. */
    switch(wsc.cont)
```

```
    {
     case 0: c0.CURS_POS = wsc.pos_arr[0];
             c1.CURS_POS = wsc.pos_arr[1];
             c2.CURS_POS = wsc.pos_arr[2];
             break;
     case 1: c0.CURS_POS = wsc.pos_arr[0];
             c2.CURS_POS = wsc.pos_arr[1];
             c3.CURS_POS = wsc.pos_arr[2];
             break;
     case 2: c0.CURS_POS = wsc.pos_arr[0];
             c2.CURS_POS = wsc.pos_arr[1];
             break;
     case 3: c2.CURS_POS = wsc.pos_arr[0];
             c3.CURS_POS = wsc.pos_arr[1];
             break;
    }
    from_stack = TRUE;
    return(wsc.cont);
 }

                              /* If list object is most dominant. */
 if (list_time > NEG_TIME)
 {
                              /* Clear the stack flag. */
                              /* Set the cursor for the proper */
                              /* container for the dom object. */
                              /* Return the dom object container. */
    from_stack = FALSE;
    switch(wlc.cont)
    {
     case 0: c0.CURS_POS = wlc.cursor_position;
             break;
     case 1: c1.CURS_POS = wlc.cursor_position;
             break;
     case 2: c2.CURS_POS = wlc.cursor_position;
             break;
    }
    return(wlc.cont);
 }

                             /* No more dom objects - we're done. */
 return(NEG_TIME);
}

/*****************************************************************/
/*** Get next item from a join.                                ***/
/*****************************************************************/

int join_from_cont0_rule0(int first_time)
{
 if (!first_time)
```

```
    goto cont0_continue;

 FOREACHC(lc1)
 {
   FOREACHC(lc2)
   {
     if ((c2.location == START) &&
         (TIMESTAMP(c0) > TIMESTAMP(c1)))
     {
        goto found_a_cont0;
     }
cont0_continue: ;
   }
 }
 return(FALSE);

found_a_cont0:
 push_ws(c0.CURS_POS, c1.CURS_POS, c2.CURS_POS, TIMESTAMP(c0), 0);
 return(TRUE);
}

int join_from_cont0_rule1(int first_time)
{
 if (c0.id == START)
    return(FALSE);

 if (!first_time)
    goto cont1_continue;

 FOREACHC(lc3)
 {
  FOREACHC(lc4)
  {
    if ((c2.eid != c3.eid) && (c2.shape == c3.shape) &&
        (c2.matched == FALSE) && (c3.matched == FALSE) &&
        (TIMESTAMP(c0) > TIMESTAMP(c2)))
    {
       goto found_a_cont1;
    }
cont1_continue: ;
  }
 }
 return(FALSE);

found_a_cont1:
 push_ws(c0.CURS_POS, c2.CURS_POS, c3.CURS_POS, TIMESTAMP(c0), 1);
 return(TRUE);
}

int join_from_cont0_rule2(int first_time)
{
```

```
 if (c0.id == START)
    return(FALSE);

 if (!first_time)
    goto cont2_continue;

 FOREACHC(lc3)
 {
   if ((c2.matched == FALSE) &&
       (TIMESTAMP(c0) > TIMESTAMP(c2)))
   {
      goto found_a_cont2;
   }
cont2_continue: ;
 }
 return(FALSE);

found_a_cont2:
 push_ws(c0.CURS_POS, c2.CURS_POS, c2.CURS_POS, TIMESTAMP(c0), 2);
 return(TRUE);
}

int join_from_cont1(int first_time)
{
 if (!first_time)
    goto cont3_continue;

 FOREACHP(lc1)
 {
   FOREACHC(lc2)
   {
     if ((c2.location == START) &&
         (TIMESTAMP(c1) > TIMESTAMP(c0)))
     {
        goto found_a_cont3;
     }
cont3_continue: ;
   }
 }
 return(FALSE);

found_a_cont3:
 push_ws(c0.CURS_POS, c1.CURS_POS, c2.CURS_POS, TIMESTAMP(c1), 0);
 return(TRUE);
}

int join_from_cont2_rule0(int first_time)
{
 if (c2.location != START)
    return(FALSE);
```

```
 if (!first_time)
     goto cont4_continue;

 FOREACHP(lc2)
 {
   FOREACHP(lc1)
   {
     if (TIMESTAMP(c2) > TIMESTAMP(c1))
     {
        goto found_a_cont4;
     }
cont4_continue: ;
   }
 }
 return(FALSE);

found_a_cont4:
 push_ws(c0.CURS_POS, c1.CURS_POS, c2.CURS_POS, TIMESTAMP(c2), 0);
 return(TRUE);
}

int join_from_cont2_rule1(int first_time)
{
 if (c2.matched != FALSE)
    return(FALSE);

 if (!first_time)
    goto cont5_continue;

 FOREACHP(lc3)
 {
   if (c0.id != START)
   {
     FOREACHC(lc4)
     {
       if ((c2.eid != c3.eid) && (c2.shape == c3.shape) &&
           (c3.matched == FALSE) &&
           (TIMESTAMP(c2) > TIMESTAMP(c0)))
       {
          goto found_a_cont5;
       }
cont5_continue: ;
     }
   }
 }
 return(FALSE);

found_a_cont5:
 push_ws(c0.CURS_POS, c2.CURS_POS, c3.CURS_POS, TIMESTAMP(c2), 1);
 return(TRUE);
}
```

```
int join_from_cont2_rule2(int first_time)
{
 if (c2.matched != FALSE)
    return(FALSE);

 if (!first_time)
    goto cont6_continue;

 FOREACHP(lc3)
 {
   if ((TIMESTAMP(c2) > TIMESTAMP(c0)) &&
       (c0.id != START))
   {
      goto found_a_cont6;
   }
cont6_continue: ;
 }
 return(FALSE);

found_a_cont6:
 push_ws(c0.CURS_POS, c2.CURS_POS, c2.CURS_POS, TIMESTAMP(c2), 2);
 return(TRUE);
}

int join_from_cont2_rule3(int first_time)
{
 if ((c2.location != BRUTE_FORCE) ||
     (c2.matched != FALSE))
    return(FALSE);

 if (!first_time)
     goto cont7_continue;

 FOREACH(c3)
 {
   if ((TIMESTAMP(c2) > TIMESTAMP(c3)) &&
       (c2.eid != c3.eid) && (c2.shape == c3.shape) &&
       (c3.location == BRUTE_FORCE) && (c3.matched == FALSE))
   {
      goto found_a_cont7;
   }
cont7_continue: ;
 }
 return(FALSE);

found_a_cont7:
 push_ws(c2.CURS_POS, c3.CURS_POS, c3.CURS_POS, TIMESTAMP(c2), 3);
 return(TRUE);
}
```

```
/*****************************************************************/
/*** Fire the rules.                                           ***/
/*****************************************************************/

void fire_rule0()
{
 c2.location = c1.color;
 push_wl(2, c2.CURS_POS, TIMESTAMP(c2));
 last_rule_fired = 0;
}

void fire_rule1()
{
 printf(" match edges %d %d", c2.eid, c3.eid);
 c2.matched = TRUE;
 push_wl(2, c2.CURS_POS, TIMESTAMP(c2));
 c3.matched = TRUE;
 push_wl(2, c3.CURS_POS, TIMESTAMP(c3));
 last_rule_fired = 1;
}

void fire_rule2()
{
 c2.location = BRUTE_FORCE;
 push_wl(2, c2.CURS_POS, TIMESTAMP(c2));
 last_rule_fired = 2;
}

void fire_rule3()
{
 printf(" match edges %d %d", c2.eid, c3.eid);
 c2.matched = TRUE;
 push_wl(2, c2.CURS_POS, TIMESTAMP(c2));
 c3.matched = TRUE;
 push_wl(2, c3.CURS_POS, TIMESTAMP(c3));
 last_rule_fired = 3;
}

/*****************************************************************/
/*** Find which rule to fire.                                  ***/
/*****************************************************************/

void check_cont0(void)
{
 int cont;

 if   (from_stack)
 {
     cont = wsc.cont;
     pop_ws();
     rule_to_fire = NEG_TIME;
```

```
     switch (cont)
     {
      case 0: if (join_from_cont0_rule0(!from_stack))
                 rule_to_fire = 0;
              break;
      case 2: if (join_from_cont0_rule2(!from_stack))
                 rule_to_fire = 2;
              break;
   }
 }
 else
 {
     pop_wl();
     if (join_from_cont0_rule0(!from_stack))
        rule_to_fire = 0;
     else if   (join_from_cont0_rule1(!from_stack))
               rule_to_fire = 1;
          else if   (join_from_cont0_rule2(!from_stack))
                    rule_to_fire = 2;
               else rule_to_fire = NEG_TIME;
 }
}

void check_cont1(void)
{
 if   (from_stack)
      pop_ws();
 else pop_wl();

 if   (join_from_cont1(!from_stack))
      rule_to_fire = 1;
 else rule_to_fire = NEG_TIME;
}

void check_cont2(void)
{
 int cont;

 if   (from_stack)
 {
      cont = wsc.cont;
      pop_ws();
      rule_to_fire = NEG_TIME;
      switch (cont)
      {
      case 0: if (join_from_cont2_rule0(!from_stack))
                 rule_to_fire = 0;
              break;
      case 1: if (join_from_cont2_rule1(!from_stack))
                 rule_to_fire = 1;
              break;
```

```
      case 2: if (join_from_cont2_rule2(!from_stack))
                 rule_to_fire = 2;
              break;
      case 3: if (join_from_cont2_rule3(!from_stack))
                 rule_to_fire = 3;
              break;
      }
 }
 else
 {
      pop_wl();
      if  (join_from_cont2_rule0(!from_stack))
          rule_to_fire = 0;
     else if   (join_from_cont2_rule1(!from_stack))
               rule_to_fire = 1;
          else if   (join_from_cont2_rule2(!from_stack))
                    rule_to_fire = 2;
               else if   (join_from_cont2_rule3(!from_stack))
                         rule_to_fire = 3;
                    else rule_to_fire = NEG_TIME;
 }
}

/*****************************************************************/
/*** Execute the production system.                            ***/
/*****************************************************************/

void execute_production_system()
{
                              /* Get next dominant object. */
                              /* While there are more objects to */
                              /* process... */
 dom_container = get_next_dom_object();
 while (dom_container >= 0)
 {
                            /* Find out which rule to fire based */
                             /* on the container of the dom obj. */
   switch (dom_container)
   {
    case 0: check_cont0();
            break;
    case 1: check_cont1();
            break;
    case 2:
    case 3: check_cont2();
            break;
   }

                                        /* Fire the proper rule. */
   switch(rule_to_fire)
   {
```

```
     case 0: fire_rule0();
             break;
     case 1: fire_rule1();
             break;
     case 2: fire_rule2();
             break;
     case 3: fire_rule3();
             break;
    }

                                    /* Get next dominant object. */
 dom_container = get_next_dom_object();
 }
}

/*************************************************************/
/*** Set up the initial data set. ***/
/*************************************************************/

void init_data(void)
{
                             /* No last rule fired. */
                             /* We're not yet done with the pgm. */
 last_rule_fired = NEG_TIME;
 done = FALSE;

 printf("OPS5c - LEAPS-based OPS5 Compiler\n");
 printf("Release 5.4 (under developement)\n");
 printf("(C) 1988,1989 D.P.Miranker, B.J.Lofaso,");
 printf(" A.Chandra Univ. of Texas at Austin\n");
}

/*************************************************************/
/*** Insert a data tuple item. ***/
/*************************************************************/

int get_token(char t[])
{
 int t_len;

 t_len = strlen(t) - 1;
 if (t[t_len] == R_PAREN)
    t[t_len] = NULL_CHAR;

 if (strcmp(t, "false") == 0)
    return FALSE;
 if (strcmp(t, "true") == 0)
    return TRUE;
 if (strcmp(t, "start") == 0)
    return START;
 if (strcmp(t, "blue") == 0)
```

```
    return BLUE;
 if (strcmp(t, “green”) == 0)
    return GREEN;
 if (strcmp(t, “white”) == 0)
    return WHITE;
 if (strcmp(t, “orange”) == 0)
    return ORANGE;
}

void insert_data_tuple(void)
{
 struct pile t_pile;
 struct piece t_piece;
 struct edge t_edge;

 char cont_name[TOKEN_LEN];                  /* Container name. */
 char field_name[TOKEN_LEN];                 /* Field name. */
 int i1, i2;
 char ch1;
 char s1[TOKEN_LEN], s2[TOKEN_LEN],
 s3[TOKEN_LEN], s4[TOKEN_LEN],
 s5[TOKEN_LEN], s6[TOKEN_LEN],
 s7[TOKEN_LEN];


                                    /* Scan the container name. */
 sscanf(t_ptr2, “%s”, cont_name);

                            /* For each container type, scan the */
                            /* line, add the data values. */
                            /* Insert into container. */
                            /* Push onto the wait list. */
                            /* Return from this insertion. */
 if (strcmp(cont_name, “pile”) == 0)
 {
    sscanf(t_ptr2, “%s %s %s”, cont_name, s1, s2);
    t_pile.id = get_token(s2);
    INS(cont0, t_pile, c0);
    push_wl(0, c0.CURS_POS, TIMESTAMP(c0));
    return;
 }
 if (strcmp(cont_name, “piece”) == 0)
 {
    sscanf(t_ptr2, “%s %s %d %s %s”, cont_name, s1, &i1, s2, s3);
    t_piece.pid = i1;
    t_piece.color = get_token(s3);
    INS(cont1, t_piece, c1);
    push_wl(1, c1.CURS_POS, TIMESTAMP(c1));
    return;
 }
 if (strcmp(cont_name, “edge”) == 0)
```

```
 {
    sscanf(t_ptr2, "%s %s %d %s %d %s %c %s %s %s %s",
    cont_name, s1, &i1, s2, &i2, s3, &ch1, s4, s5, s6, s7);
    t_edge.pid = i1;
    t_edge.eid = i2;
    t_edge.shape = ch1 - 'A' + A;
    t_edge.matched = get_token(s5);
    t_edge.location = get_token(s7);
    INS(cont2, t_edge, c2);
    push_wl(2, c2.CURS_POS, TIMESTAMP(c2));
    return;
 }
}

/*****************************************************************/
/*** Main routine for the program.                             ***/
/*****************************************************************/

void main(void)
{
                              /* Set initial data. */
                              /* While there's more input... */
 init_data();
 while (!done)
 {
                              /* Print the prompt. */
                              /* Get a line of data. */
                              /* Find the open paren (fake LISP). */
                              /* If there is an open paren... */
   printf("\nTop Level> ");
   gets(inp_line);
   t_ptr = strchr(inp_line, L_PAREN);
   if (t_ptr != NULL)
   {
                              /* Skip to the next char. */
                              /* Find the end of the token. */
                              /* Clear out token. */
                              /* Copy the token. */
      t_ptr2 = ++t_ptr;
      while (((char) *t_ptr2 != SPACE) &&
             ((char) *t_ptr2 != R_PAREN))
      {
        t_ptr2++;
      }
      memset(token, 0, TOKEN_LEN);
      strncpy(token, t_ptr, (t_ptr2 - t_ptr));

                               /* 'exit' means leave the program. */
                               /* 'run' means execute the system. */
                               /* 'make' means load a data item. */
      if   (strcmp(token, EXIT) == 0)
```

```
        {
             printf("Exiting system.\n");
             done = TRUE;
        }
        else if (strcmp(token, RUN) == 0)
             {
                 execute_production_system();
                 fprintf(stderr, "No instances in conflict set.\n");
             }
             else if (strcmp(token, MAKE) == 0)
                     insert_data_tuple();
    }
  }
}
```

Appendix C

Quicksort sample code

This appendix lists both the BSD and Predator quicksort code:

BSD Quicksort

```
void qsort(base, n, size, compar)
char*base;
int n;
int size;
int (*compar)();
{
  register char c, *i, *j, *lo, *hi;
  char *min, *max;
  if (n <= 1)
     return;
  qsz = size;
  qcmp = compar;
  thresh = qsz * THRESH;
  mthresh = qsz * MTHRESH;
  max = base + n * qsz;
  if (n >= THRESH) {
     qst(base, max);
     hi = base + thresh;
  } else {
     hi = max;
  }
  for (j = lo = base; (lo += qsz) < hi;)
    if (qcmp(j, lo) > 0)
       j = lo;
  if (j != base) {
     for (i = base, hi = base + qsz; i < hi; ) {
       c = *j;
       j++ = *i;
       *i++ = c;
    }
  }
```

```
  for (min = base; (hi = min += qsz) < max; ) {
    while (qcmp(hi -= qsz, min) > 0)
      if ((hi += qsz) != min) {
         for (lo = min + qsz; --lo >= min; ) {
           c = *lo;
           for (i = j = lo; (j -= qsz) >= hi; i = j)
             *i = *j;
           *i = c;
        }
      }
  }
}

static qst(base, max)
char *base, *max;
{
  register char c, *i, *j, *jj;
  register int ii;
  char *mid, *tmp;
  int lo, hi;

  lo = max - base;/
  do{
    mid = i = base + qsz * ((lo / qsz) >> 1);
    if (lo >= mthresh) {
       j = (qcmp((jj = base), i) > 0 ? jj : i);
       if (qcmp(j, (tmp = max - qsz)) > 0) {
          j = (j == jj ? i : jj);
          if (qcmp(j, tmp) < 0)
          j = tmp;
      }
      if (j != i) {
         ii = qsz;
         do{
           c = *i;
           *i++ = *j;
           *j++ = c;
         } while (--ii);
      }
      for (i = base, j = max - qsz; ; ) {
        while (i < mid && qcmp(i, mid) <= 0)
          i += qsz;
        while (j > mid) {
          if (qcmp(mid, j) <= 0) {
             j -= qsz;
             continue;
          }
        tmp = i + qsz;/
        if (i == mid) {
           mid = jj = j;
        } else {
```

```
            jj = j;
            j -= qsz;
          }
          goto swap;
        }
        if (i == mid) {
           break;
        } else {
           jj = mid;
           tmp = mid = i;/
           j -= qsz;
        }

swap:
        ii = qsz;
        do{
          c = *i;
          *i++ = *jj;
          *jj++ = c;
        } while (--ii);
        i = tmp;
      }
      i = (j = mid) + qsz;
      if ((lo = j - base) <= (hi = max - i)) {
         if (lo >= thresh)
         qst(base, j);
          base = i;
         lo = hi;
      } else {
         if (hi >= thresh)
            qst(i, max);
         max = j;
      }
    } while (lo >= thresh);
}
```

Predator quicksort

```
void quick(start_pos, end_pos, comp, size)
register CURS_POS start_pos;
register CURS_POS end_pos;
COMP comp;
int size
{
  register CURS_POS save;
  register int left_size, right_size;
  short select_size;
  short loop;
```

```
short t_short;
short ts2, ts3;

if (start_pos == end_pos)       /* If zero size, return.      */
   return;

asc.CURS_POS = start_pos;       /* Set start position.        */

if (size <= 5)                  /* Base case - bubble sort.   */
{
  pivot.CURS_POS = start_pos;
  ADV(pivot);
  save = pivot.CURS_POS;

                                /* Outer loop.                */
  for (select_size = size - 1; select_size > 0;
         select_size--)
  {
                                /* Inner loop.                */
                                /*  If out of order, save it. */
                                /* Continue if not done.      */
    for (loop = 0; loop < select_size; loop++)
    {
      t_short = ((*comp)(asc.CURS_POS,
                                   pivot.CURS_POS));
      if (t_short < 0)
      {
         asc.CURS_POS = pivot.CURS_POS;
      }
      if (loop < select_size - 1)
      {
         ADV(pivot);
      }
    }

                                /* If need to swap, do it.    */
                                /* Set up for next iteration. */
    if (asc.CURS_POS != pivot.CURS_POS)
       SWAP(asc, pivot);
    asc.CURS_POS = start_pos;
    pivot.CURS_POS = save;
  }
  return;                       /* End base case.             */
}

                                /* Pick a pivot of 2nd item.  */
pivot.CURS_POS = start_pos;
ADV(pivot);

                                /* Loop through for pivot     */
```

```
for (loop = 0; loop < (size - 1); loop++)
{
                         /* Compare first two...              */
                         /* If not the same...                */
                         /*  Move if conveniently near end.   */
     t_short = ((*comp)(asc.CURS_POS,
                        pivot.CURS_POS));
     if (t_short != 0)
     {
        if (loop == (size - 2))
        {
           if (t_short > 0)
           {
              pivot.CURS_POS = asc.CURS_POS;
           }
           break;
     }
                         /* Look at last item.                */
                         /* Compare three items.              */
                         /* Take median for pivot.            */
     dsc.CURS_POS = end_pos;
     ts2 = ((*comp)(pivot.CURS_POS,
                    dsc.CURS_POS));
     ts3 = ((*comp)(asc.CURS_POS,
                  dsc.CURS_POS));
     if   (ts3 < 0)
     {
          if   (ts2 > 0)
          {
               pivot.CURS_POS = dsc.CURS_POS;
          }
          else if (t_short > 0)
               {
                  pivot.CURS_POS = asc.CURS_POS;
               }
          }
          else
          {
               if   (ts2 < 0)
               {
                    pivot.CURS_POS = dsc.CURS_POS;
               }
               else if (t_short < 0)
                    {
                       pivot.CURS_POS =
                      asc.CURS_POS;
                    }
          }
          break;
```

```
  }
  asc.CURS_POS = pivot.CURS_POS;
  ADV(pivot);
}

                              /* Return if no work needed.      */
if (loop == (size - 1))
   return;

                              /* Main loop. Set the sizes.      */
                              /* Set starting cursors.          */
                              /* While there's more to search.  */
                              /*  Scan right until find one     */
                              /*   bigger than pivot.           */
                              /*  Scan left until find one      */
                              /*   smaller than pivot.          */
                              /*  If past center, swap two items*/
left_size = right_size = 1;
asc.CURS_POS = start_pos;
dsc.CURS_POS = end_pos;
while ((left_size + right_size) <= size)
{
  while (((*comp)(asc.CURS_POS,
                  pivot.CURS_POS)) < 0)
  {
    left_size++;
    ADV(asc);
  }
  while (((*comp)(dsc.CURS_POS,
                  pivot.CURS_POS)) >= 0)
  {
    right_size++;
    REV(dsc);
  }
  if ((left_size + right_size) <= size)
  {
      if   (asc.CURS_POS == pivot.CURS_POS)
           pivot.CURS_POS = dsc.CURS_POS;
      SWAP(asc, dsc);
  }
}

                              /* Save position for second quick */
                              /* Swap pivot to middle.          */
                              /* If partition should be sorted, */
                              /*  do it.                        */
                              /* Same for other partition.      */
save = dsc.CURS_POS;
SWAP(asc, pivot);
if (right_size > 3)
{
```

```
      ADV(asc);
      quick(asc.CURS_POS, end_pos, comp,
               right_size - 2);
   }
   if (left_size > 1)
      quick(start_pos, save, comp, left_size - 1);
}
```

Appendix D

Benchmark source code

This appendix lists the benchmarking program (array version) for the Predator spelling checker benchmark:

```
#include <stdio.h>

#define EXIT_ERROR 1
#define EXIT_NORMAL 0
#define MAX_WORD_SIZE 32

struct word_elem
{
 char wrd[32];
};

SCHEMA dict_struct ON ELEMENT word_elem = array[25000];
dict_struct the_dict;
CURSOR c ON the_dict;

SCHEMA doc_struct ON ELEMENT word_elem = array[10000];
doc_struct document;
CURSOR c2 ON document;

void read_in_dict ()
{
 FILE *dict_file;
 int word_count = 0;
 struct word_elem t_word;
 char *tc;
 int i;

 printf ("Initializing");
 dict_file = fopen ("words", "r");
 if (dict_file == NULL)
 {
    printf ("\nCould not open dictionary.\n");
    exit(EXIT_ERROR);
```

```
 }

 while (1)
 {
  tc = t_word.wrd;
  i = 0;
  do
    *tc = getc (dict_file);
  while (!feof (dict_file) && i++ < MAX_WORD_SIZE && *tc != '\n' &&
         tc++);
  *tc = '\0';
  if (i == 0)
     break;

  INS (the_dict, t_word, c);

  if ((++word_count % 100) == 1)
  {
     printf (".");
     fflush (stdout);
  }
 }
 fclose (dict_file);
 printf ("\nThe dictionary contains %d words.\n", word_count);
}

void process_input_file ()
{
 struct word_elem t_word;
 char *t_ptr;
 char t_char;

 while (!feof (stdin))
 {
  t_ptr = t_word.wrd;
  *t_ptr = '\0';

  while (!feof (stdin))
  {
   t_char = fgetc (stdin);
   if (isalpha (t_char))
      break;
  }

  if (feof(stdin))
     return;

  while (!feof (stdin))
  {
   if   (isalpha (t_char))
        *t_ptr++ = t_char;
```

```
   else
   {
        *t_ptr = '\0';
        break;
   }
   t_char = fgetc (stdin);
  }

  for (t_ptr = t_word.wrd; *t_ptr; t_ptr++)
   if (isupper (*t_ptr))
      *t_ptr = tolower (*t_ptr);

  RESET_CURSOR (c2, CONT_START);
  FIND (c2, strcmp (wrd, t_word.wrd) == 0);
  if (LAST_CURS_OP (c2) == EOR)
  {
     INS (document, t_word, c2);
  }
 }
}

void print_output_data ()
{
 FOREACH (c2)
 {
  RESET_CURSOR (c, CONT_START);
  FIND (c, strcmp (c.wrd, c2.wrd) == 0);
  if (LAST_CURS_OP (c) == EOR)
     printf (" %s", c2.wrd);
 }
 printf ("\n");
}

void main (void)
{
 read_in_dict ();
 process_input_file ();
 print_output_data ();
 exit (EXIT_NORMAL);
}
```

Appendix E

Supermarket macros

This appendix lists the macros necessary to implement the operations for the supermarket example described in Section 1.2. Note that the following code examples could be placed directly inline in the source program)[1]:

- A customer whose name is **cust_name** leaves the middle of a line to do more shopping:

```
MACRO leave_line(container, cust_name)
{
  RESET(container.cursor, CONST_START);
  FIND(container.cursor, (STRING) l_name == cust_name);
  DELETE(container.cursor);
};
```

- All customers in line B with 10 or fewer items moves to line A:

```
MACRO express_line(lineA, lineB)
{
  FOREACH(lineB.cursor)
  {
    if (num_items <= 10)
      {
        GETREC(lineB.cursor, temp_customer);
        INSERT(lineA, temp_customer, t_cursor);
        DELETE(lineB.cursor);
    }
  }
```

- A new customer moves in line in front of customer **cust_name**:

---

1. The need for composite functions has been recognized previously [Boo87, McN86a, McN86b]. My macros are a slightly modified version of the same concept.

```
MACRO add_customer(container, cust_name, new_customer)
{
  FIND(container.cursor, (STRING) l_name == “Smith”);
  INSERT(container, new_customer,container.cursor, BEFORE_CURSOR);
};
```

- Every third customer in a line puts a candy bar in their basket:

```
MACRO add_third(container, item)
{
  count = 0;
  FOREACH(container.cursor)
  {
    if (count % 3 == 0)
    {
       candy_bought = TRUE;
       num_items++;
    }
  }
};
```

Appendix F

Functional templates

This appendix contains the functional templates used by Predator for the basic set of primitive functions of the **DS** realm. Sections which are underlined are those which are added by the layers The comments in the templates below list sample layers that might implement the snippet function, or sample actions those layers might perform in the snippet function.

F.1 Snippet functions

The following table details the snippet functions implemented in the Predator compiler:

Table F.1: Snippet functions

Snippet Function	**Description**
special_processing	Allows for actions to occur before or after the primitive function such as element locking and before actions
set_to_first_item	Set the cursor to the first element in the container (based on the cursor's scanning layer)
past_end_of_container?	Is the cursor past the last element of the container?
disqualification_predicate	Boolean condition(s) which would disqualify this element from being used (such as delflag)
advance_cursor	Move the cursor to the next element in the container
set_status_flag	Set the status flag for the cursor based on the results of the last advance_cursor operation
additional_fields	Set additional fields (such as segmentation and links)
set_cursor/allocate	Allocate memory for the element, and set the cursor to point to the new element
copy_record	Copy the record into the allocated storage
init_special_flds	Intialize special fields for the insertion (such as segmentation)

Table F.1: Snippet functions

Snippet Function	Description
attach_elem	Attach the element to the data structure (such as linked lists, binary trees, and indexes
move_elem_within_cont	Move the updated element to its new location in the data structure(s) defined by the layers
set_del_fields	Set the fields for deletion

F.2 Delete

```
special processing();     /* Before actions, concurrency...     */
set del fields();         /* Set delflag field, unlink from     */
                          /*  lists, free memory.               */
special processing();     /* After actions, concurrency...      */
```

F.3 Reset

```
special processing();     /* Before actions, concurrency...     */
set to first item();      /* Scanning layer implements this.    */
while (!(past end of container?()) &&
       !(cursor's predicate) &&
       (disqualification predicate()))
{
  advance cursor();       /* Scanning layer.                    */
}
set status flag();        /* Did we find an item?               */
additional fields();      /* Such as additional segment cursors */
special processing();     /* After actions, concurrency...      */
```

F.4 Advance/Reverse

```
special processing();     /* Before actions, concurrency...     */
advance cursor();         /* Scanning layer.                    */
while (!(past end of container?()) &&
       !(cursor's predicate) &&
       (disqualification predicate()))
{
  advance cursor();       /* Scanning layer.                    */
}
set status flag();        /* Did we find an item?               */
additional fields();      /* Such as additional segment cursors */
special processing();     /* After actions, concurrency...      */
```

F.5 Foreach

```
special processing();     /* Before actions, concurrency...     */
set to first item();      /* Scanning layer implements this.    */
while !(past end of container?())
{
  if ((cursor's predicate) &&
      !(disqualification predicate()))
  {
     /* Code fragment for FOREACH goes here */
  }
  advance cursor();       /* Scanning layer.                    */
}
special processing();     /* After actions, concurrency...      */
```

F.6 Find

```
special processing();     /* Before actions, concurrency...     */
while !(past end of container?())
{
  if ((cursor's predicate) && (find predicate) &&
      !(disqualification predicate()))
  {
     break;
  }
  advance cursor();       /* Scanning layer.                    */
}
set status flag();        /* Did we find an item?               */
additional fields();      /* Such as additional segment cursors */
special processing();     /* After actions, concurrency...      */
```

F.7 Insert

```
special processing();     /* Before actions, concurrency...     */
set cursor/allocate();    /* Allocate memory, move to next array*/
                          /*  element, get avail item...        */
copy record();            /* Terminal layer.                    */
init special flds();      /* Delflag, segment, size...          */
attach elem();            /* Index, predicate index, links...   */
special processing();     /* After actions, concurrency...      */
```

F.8 Update

```
special processing();      /* Before actions, concurrency...     */
update field;              /* Done by compiler.                  */
move elem within cont();   /* Only for index, ordered, link layers/
special processing();      /* After actions, concurrency...      */
```

F.9 Others

There are other primitive functions (such as swap, getrec) whose templates are nothing more than a single call to a snippet function, surrounded by two **<special processing>** functions.

Vita

Martin J. Sirkin was born in Boston, Massachusetts, and graduated from Sharon High School in 1979. He received his Bachelor of Science degree from the California Institute of Technology, in Pasadena, in 1984 and Master of Science from the University of Washington, in Seattle, in 1988. He completed his doctoral studies at the University of Washington in 1994.

Marty has worked for IBM in Austin, Texas since 1989. He is currently a staff programmer in the Personal Systems Products division. His work in databases, graphical user interfaces, and distributed computing, has resulted in nineteen technical disclosures and four patents filed on behalf of IBM.

The following is a list of Marty's previous publications:

1. M. Sirkin, Configuring Remote Data Services (pt 1), *IBM Personal Systems Developer*, 4: 115-123, Fall 1990.

2. M. Sirkin, Installing and Configuring the DOS Database Requester, *IBM Personal Systems Technical Solutions*, 2: 47-50, 1991.

3. M. Sirkin, Configuring Remote Data Services (pt 2), *IBM Personal Systems Developer*, 1: 94-102, Winter 1991.

4. D. Batory, V. Singhal, and M. Sirkin, Implementing a domain model for data structures, *International Journal of Software Engineering and Knowledge Engineering*, 2(3): 375-402, September 1992.

5. M. Sirkin, D. Batory, and V. Singhal, Software components in a data structure precompiler, In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore MD, 437-446, May 1993.

6. J. Thomas, D. Batory, V. Singhal, and M. Sirkin, A Scalable Approach to Software Libraries, In *Proceedings of the 6th Annual Workshop on Software Reuse*, Owego, NY, November 1993.

7. D. Batory, V. Singhal, J. Thomas, and M. Sirkin, Scalable Software Libraries, In *Proceedings of the ACM SigSoft '93 Conference*, Los Angeles, CA, December 1993.