International Conference on Software Engineering (ICSE) (Baltimore, MD), May 1993, pages 437-446

Software Components in a Data Structure Precompiler[1]

Marty Sirkin, Don Batory, Vivek Singhal
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{marty, batory, singhal}@cs.utexas.edu

Abstract

PREDATOR is a data structure precompiler that generates efficient code for maintaining and querying complex data structures. It embodies a novel component reuse technology that transcends traditional generic data types. In this paper, we explain the concepts of our work and our prototype system. We show how complex data structures can be specified as compositions of software building blocks, and present performance results that compare PREDATOR output to hand-optimized programs.

1: Introduction

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 disproportionally large fraction of a programmer's time. A *data structure compiler* is a suite of tools that reduces the burden of programming data structures. There have been several attempts to produce such compilers. Three examples are [10, 19, 25]. In general, however, data structure compilers have not achieved a broad level of acceptance. The reasons include inadequate performance, unnecessary complexity, host language restrictions, and limited scope.

Eliminating the drudgery of programming data structures is clearly an important problem. We believe the solution rests on a software component technology that is defined by a combination of concepts from databases, compilers, transformation systems, and domain modelling. While none of these concepts are new, we are presenting a unique combination that yields a technology for assembling complex data structures from plug-compatible components.

Two goals of a data structure compiler should be:

1. To generate efficient code, i.e. within 10% of highly tuned and hand optimized code.
2. To allow programs to be easily written in a data structure independent manner. This would allow data structures to be changed without modifying the application program.

When programs employ component technologies, there are two distinct phases of software development: *component creation* and *application writing*. Component creation involves the definition of the interface and the implementation of software components. In the application writing phase, components are combined with customized (application-specific) code to form the completed program. Occasionally, a programmer is forced to implement new components, thus mixing the two phases. *Our research aims to simplify component creation and reduce the necessity of implementing new components.* This division of specification and tools from application programming is not unique [26, 23]. Our research concentrates more on programmer productivity and code efficiency than on program evolution and maintenance.

Our project is called PREDATOR, which is a (misspelled) acronym for PREcompiler for DAta sTRuctures. In this paper, we motivate the need for PREDATOR by exposing important limitations of traditional parameterized types, the central concept upon which all existing data structure compilers are built (to our knowledge). We then discuss the required abstractions and how they overcome the limitations that we identify.

2: Traditional Parameterized Types

Existing data structure compilers accomplish software reuse through *traditional parameterized types* (TPTs), e.g., C++ templates [24] and ADA generics [13]. A TPT is a generic type whose parametric instantiations define a family of related types. The classic example is the array, which can be instantiated to produce arrays of integers, arrays of

[1] This research supported in part by grants from Texas Instruments, IBM, and Digital Equipment Corporation.

strings, etc. TPTs can be instantiated by primitive types or by other instantiated TPTs. For example, Figure 1 depicts a linked list TPT that has been instantiated with a binary tree TPT, where each node of the list is the root of a distinct binary tree.

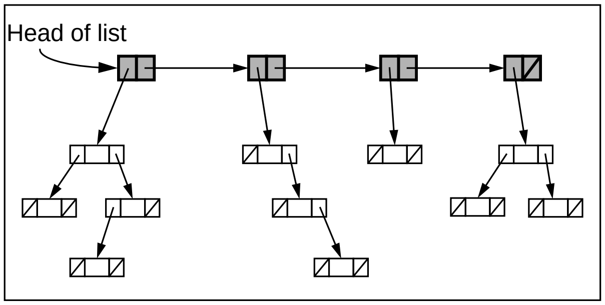


Figure 1: Traditional parameterized type: a list of trees

There is no consensus on how software reuse can be achieved [5]. In our opinion, there are three requirements for successful software reuse. First, software components must be *designed* to be plug-compatible and interchangeable. Second, programming languages should provide appropriate *features* for implementing such designs. And third, these designs and programming language features should not incur exorbitant *performance* penalties.

TPTs and their conventional use are insufficient to achieve these requirements. The following discussion explains why:

1. **Difficulty of specialization (conceptual limitation)**. A TPT offers operations that its author believes are adequate for a wide variety of applications. A queue TPT, for example, would likely provide the operations *enqueue, dequeue, is_empty* and *is_full*. However, in the context of a specific application, it occurs frequently that additional operations, unforeseen by the TPT author, are needed. Suppose that it is necessary to delete items located in the interior of a queue. Using only the given operations, one must dequeue each item, check to see if it is the requested item, and enqueue it if it is not. Typically the end of the queue is determined by either knowing the size of the queue or by using an end-of-queue marker. Clearly this is both inefficient and awkward. The situation is no better for stacks.[2]

   Specialization is the process of modifying a type to provide a customized interface for an application. Specialization of TPTs poses some difficult problems [6, 9, 20, 24]:

   - Attempting to provide an exhaustive TPT interface actually discourages reuse. Programmers are intimidated by complex interfaces and would rather design their own.
   - Given TPT source code, a programmer could attempt to integrate his own changes. However, this essentially nullifies the productivity advantages of TPTs, as programmers must now understand someone else's code in order to write and debug their extensions.
   - New operations can be defined in terms of existing operations. However, it may not be possible to efficiently implement new operations. (Recall the queue example).

   These problems force the application writer to choose from these unattractive alternatives: not using TPTs at all, modifying TPT source code, or accepting performance inefficiencies. Even with the addition of object-oriented inheritance, these problems remain.

2. **Complex compositions (conceptual limitation)**. It is widely believed that TPTs are the appropriate abstraction for encapsulating primitive data structures [6, 17, 18]. More complex structures, such as the one depicted in Figure 1, are created through TPT compositions.

   Software component libraries are unlikely to provide implementations for many data structures used in practice; rather, the idea is to form complex structures through composition of available TPTs. 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 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.

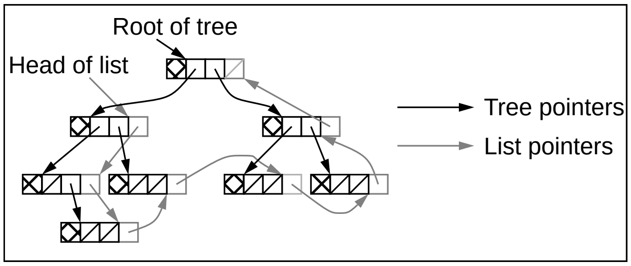


Figure 2: A binary tree/linked list data structure

[2] While this example seems contrived, it really is not. A supermarket checkout line is a queue. A common event is for a customer to leave the middle of a queue when he realizes he needs another item.

It is important to recognize that the structure in Figure 2 *cannot* be created by a parametric instantiation of the tree and linked list TPTs; a list of binary trees (Figure 1) and a binary tree of lists are definitely not equivalent to the structure of Figure 2.

The structure of Figure 2 can be created using approaches such as composing TPTs via multiple inheritance. However, application programmers must write additional "glue" code to achieve the correct semantics in the resulting "composite" TPT. Other languages such as SELF [8] provide more sophisticated inheritance mechanisms, but still suffer from generality (lack of knowledge about the data structures domain) and hence are not particularly efficient in creating merged, efficient components. *Mapping* TPTs [6] can be used to simulate (but not match identically) the structure of Figure 2, but "glue" code still needs to be written. Moreover, the resulting composite TPT incurs an additional performance penalty because mapping TPTs introduce extra pointer indirections.

Component composition is a basic operation of a data structure compiler. As stated earlier, a fundamental goal of data structure compilers is to automatically generate the "glue" code without incurring performance penalties. Thus, we feel that TPTs without compilation and domain-specific knowledge are not sufficient to define complex data structures.

3. **Type transformations (conceptual limitation).** 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 details are visible. For example, a linked list TPT adds **`previous`** and **`next`** pointers to the records that it stores.

   Inheritance is a simple but common example of 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 field or operation additions. For example, a compression transformation maps an abstract record type to a completely different type in which the fields of the original record type are no longer identifiable. A partitioning transformation, which divides records into (say) 100 byte segments, may not partition abstract records cleanly along field boundaries. Only after the abstract record is reassembled from its partitions can the original fields (and their contents) be accessible. These are only two of many possible type transformations that occur in data structures. Because TPTs rely on mechanisms such as inheritance to transform types, TPTs cannot express certain type transformations, and hence cannot express important classes of data structures.

4. **Field parameters (conceptual limitation)**. To our knowledge, parameterized types of existing programming languages may only be instantiated with constants, functions, and data types. Certain data structures may require TPTs to have fields as parameters. An ordered linked list TPT, for example, would be parameterized by the record type to be stored, and the field within that record type on which the list elements are to be sorted. While the concept of TPTs does not preclude field parameters, the need for field parameters has not been fully appreciated.

5. **Ad hoc interfaces (design limitation)**. Most TPTs have unique interfaces. This means that the TPT interface for linked lists is (typically) different from that of arrays, binary trees, etc. When an application program is written using a specific TPT, it is difficult to change the underlying data structure (TPT) without triggering a substantial rewrite.

   It is often the case that the best choice of data structures for an application can only be made late in the development process. If TPTs have unique interfaces, then it may be too expensive to retrofit a better suited data structure into existing code. A basic goal of a data structure compiler is to support the interchangeability of data structures without impacting program correctness. This can be accomplished by carefully designing standardized interfaces for TPTs. Poorly designed TPTs or TPTs with incompatible interfaces make the task of a data structure compiler difficult, if not impossible. In practice, we note that existing component libraries [6, 7, 16] tend *not* to provide the same interface for all components.

6. **Code efficiency (implementation limitation)**. A standard technique of implementing TPTs is to compile the code for each TPT component separately, where references and manipulations of generic objects are performed via pointers [13]. This introduces unnecessary runtime overhead.

   Another technique of TPT implementation is macro expansion [24]. Macro expansion by itself is not always sufficient to provide efficient code, because context information is not considered when optimizing the code which results from expanding TPT compositions.

In current data structure compilers, TPTs provide a useful framework for describing reusable software components. However, for the reasons outlined above, TPTs have conceptual, design, and implementation barriers which preclude the creation of practical data structure compilers.

3: Overcoming TPT Limitations

As mentioned in the introduction, our research is based on a combination of ideas from databases, compilers, transformation systems, and domain modelling. This section describes the basic ideas underlying our work and explains how they overcome the TPT limitations previously identified.

3.1: Design Limitation: Ad Hoc Interfaces

Many common data structures are implementations of a rather simple abstraction: a *container* of objects; binary trees, lists, and arrays are examples. The choice of container implementation is often made for performance reasons. There are many possible interfaces to containers; most expose the container's implementation. However, research on relational databases and persistent object stores have identified interfaces that do not expose a container's implementation [14, 15]. Such interfaces are ideal for use in data structure compilers because they promote interchangeable data structure components.[3]

To simplify the design of our data structure compiler, we have carefully selected an interface that is sufficiently simple that could be shared by all data structures. Any data structures which provide additional functionality can provide more functions to the interface, if need be. This interface does not expose data structure implementations, and thus permits one component to be swapped with another. Exchanging data structure components makes tuning of application programs much easier.

Consider the example of Figure 2. Suppose the records that are stored in this container are of type **customer**. In PREDATOR syntax, this data structure would be declared as:

```
CONTAINER tree_list ON ELEMENT customer
     = bintree(list(malloc));
```

The type **tree_list** stores **customer** records in a container that is defined by the composition of the **bintree**, **list**, and **malloc** data structures. **customer** records are stored in a binary tree. The nodes of the binary tree are chained together on a linked list, and list nodes are dynamically allocated on a heap. The abstract transformation model that underlies components and their composition is discussed in Section 3.3.

Our container interface largely reflects work on embedded relational languages and persistent languages. Iterations over subsets of objects in a container are accomplished through the use of iterators or *cursors* [13, 14, 1, 6]. Cursors provide SQL-like *select* capabilities where programmers declaratively specify via predicates the records (or objects) of a container that they want to retrieve [11]. Table 1 lists some of the operations of our container interface.

In PREDATOR, cursors and containers are first-class objects. They may be saved in variables, passed to functions, and stored in containers.

3.2: Conceptual Limitation: Difficulty of Specialization

The interface described in Table 1 was the result of a *domain modelling* [21] effort. This interface reflects the operations that can be performed on all data structures. Moreover, the generality and practicality of this interface has been substantiated by twenty five years of research in databases, because our container interface is virtually identical to the programming language interface for relations in relational databases [14].

We stated in Section 2 that TPT authors cannot envision all specializations. Yet, the interface that we have chosen has historically shown to be general enough to permit the definition of any other container interface. The programming language Pascal/R [22], for example, allowed users to customize the interface to relations by letting them place their own abstract data type (ADT) interfaces on top of relations and to implement ADT operations as calls to relational operators. The power of relations (containers) still remained, but a customized interface would be used in place of a relational interface.

Given our container interface, it is straightforward to define specialized interfaces to containers. For example, while it is unusual to envision stacks with a relational interface, it is easy to define the **push** and **pop** operations in terms of relational operations, as 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);
  }
};
```

[3] Note that typical database systems offer multiple container implementations which can be either selected by users or chosen at runtime by query optimizers. Databases are classical examples of systems that have successfully exploited plug-compatible implementations of containers [4].

Function Call	**Meaning**
CURSOR(k, [p [, o]])	Create and return a cursor over container *k*. The cursor can be positioned only on objects that satisfy predicate *p* and the selected objects are returned in order *o*. Both predicate *p* and order *o* are optional.
RESET(c, l)	Repositions cursor *c* either to the start of the container or to the end (based on the *l* argument).
ADVANCE(c)	Repositions cursor *c* on the next qualified object in *c*'s container. A status code is set in the cursor to OK 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.
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 data structure semantics determine the positioning of the new object.
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*
REF(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
GETREC(c, o)	Retrieve the object referenced by cursor *c* and place it into the buffer specified by *o*.
ADDRESS(c)	Return the location of the object referenced by cursor *c*.
POSITION(c, a)	Position cursor *c* on the object with location *a*.
SWAP(c1, c2)	Swap the objects referenced by cursors *c1* and *c2*. Both cursors are referencing objects in the same container.

Table 1: Partial list of primitive functions

3.3: Conceptual Limitation: Complex Compositions

We said earlier that data structures can be modelled by type transformations. In this section, we explain the model in more detail and focus on compositions of transformations. As we will see, a data structure component corresponds to an implementation of a type transformation.

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 listed in Table 1. Because it is not known how **c** is implemented, **p** is data structure generic – i.e., it is not dependent on any implementation of **c**.

The application of a transformation τ introduces data structure implementation detail. When applied to **c**, the result is container **c′**. Concomitantly, **p** must be transformed into a program **p′** that operates on **c′** and preserves the semantics of **p(c)**. Thus, applying a data structure transformation τ as a (possibly partial) implementation of **c** transforms **p(c)** to **p′(c′)**.

It follows that a data structure component (building block) for containers is a pair of functions (τ_κ: **C**⟶**C**, τ_π: **P**⟶**P**) where **C** is the domain of containers and **P** is the domain of programs. τ_κ is a *container mapping function* which transforms an abstract container **c** into a concrete (or less abstract) container **c′**. τ_π is a *program mapping function* which transforms a program **p** into a corresponding program **p′**.

Consider a component for unordered lists (**LIST**$_\kappa$: **C**⟶**C**, **LIST**$_\pi$: **P**⟶**P**). **LIST**$_\kappa$ is a container mapping function. It links together all objects of a container onto an unordered list. Figure 3a shows a container **c** with six objects. Figure 3b shows the resulting container **LIST**$_\kappa$(**c**). This container has exactly the same objects as **c**, with the addition that each object has a **next** attribute. The container itself is augmented with the attribute **head** to reference the head of the list.[4]

LIST$_\pi$: **P**⟶**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 link of the object onto a list. That is, the operation:

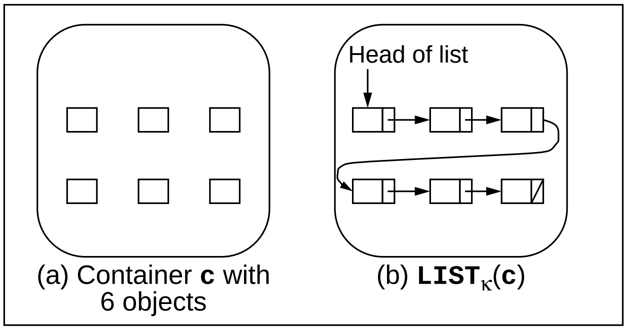
```
INSERT(c, obj, curs);
```

of program **p** is transformed into:

```
INSERT(c’, obj, curs);
UPDATE(curs, next, c’.head);
c’.head = ADDRESS(curs);
```

of program **p’**. Writing transformations for other cursor operations on **c** is straightforward.

Now consider the component for binary trees: (**BINTREE**$_\kappa$: **C**,**A**⟶**C**, **BINTREE**$_\pi$: **P**,**A**⟶**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 container are linked together onto a binary tree. Each object in **c** is transformed by the addition of two fields (**left** and **right**). The container itself is augmented with the attribute **root** to reference the root of the binary tree. Figure 4 shows the mapping of **c** to **BINTREE**$_\kappa$(**c**, **a**) where **a** is an attribute of the objects in **c.**



(a) Container **c** with 6 objects

(b) **LIST**$_\kappa$(**c**)

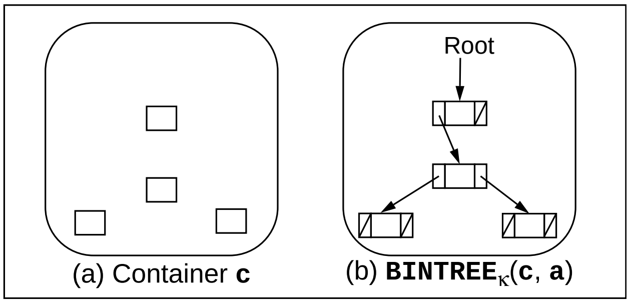
Figure 3: A simple container transformation

BINTREE$_\pi$ is the corresponding program mapping function. It transforms operations on **c** to operations on **BINTREE**$_\kappa$(**c**, **a**). For example, inserting an object into **c** is mapped to inserting the object into **BINTREE**$_\kappa$(**c**, **a**) followed by linking of 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 data structure transformations. This means that many different combinations of components are possible; each yields a different data structure and its support algorithms.

The example of Figure 2 depicts an implementation of container **c** that is implemented by a composition of the binary tree and list components: **LIST**$_\kappa$(**BINTREE**$_\kappa$(**c**)). Each object of **c** is augmented with the binary tree fields **left** and **right**, and then is augmented with the linked list field **next**. The resulting program **LIST**$_\pi$(**BINTREE**$_\pi$(**p**)) transforms an object insertion in **c** into an insertion into the container **LIST**$_\kappa$(**BINTREE**$_\kappa$(**c**)), a link of the object onto the binary tree, and then a link of the object onto the list.

Every component is parameterized by the container and program that it is to map. We call these components *non-traditional parameterized types* (NPTs). A container TPT is a composition of NPTs; NPTs are the software primitives from which an enormous class of container TPTs can be built.



(a) Container **c**

(b) **BINTREE**$_\kappa$(**c**, **a**)

Figure 4: Binary tree transformation

Programmers can choose from a large number of NPTs. Besides unordered lists and binary trees, there are NPTs for arrays, AVL trees, data compression (which transforms a container of uncompressed objects into a container of compressed objects), persistence (which transforms a container of objects in main memory into a container of objects on disk), indexing (which transforms a container of non-indexed objects into a container of indexed objects), client-server (which transforms a container of objects that appear to be stored locally into a container of objects stored remotely), and so on.

In addition to PREDATOR, another prototype implementation of these ideas is the Genesis extensible database management system. Genesis can produce a customized DBMS of 70K lines by assembling prefabricated components [3, 4].

3.4: Conceptual Limitation: Type Transformations

Recall that there are type transformations which do not add fields and operations to data types. Two examples are **segment** and **server**. **segment** partitions a data type

[4] 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$_\kappa$ mapping.

along a designated field boundary; **`server`** stores instances of a data type on a remote machine.

Consider the following data type and transformation composition:

```
struct base_elem
{
  char name[30];
  int age;
  int height;
  BYTE image[1000000];
};
```

```
CONTAINER test_cont ON ELEMENT base_elem
     = segment(dlist(server(“m”),
               height, dlist(server(“m”)));
```

The benefit of this particular NPT composition is efficiency. If a program does not access the **`image`** field, then only the **`name`**, **`age`**, and **`height`** fields are transmitted from the remote server “m”. The data types that result from this transformation are shown below, and are automatically generated by PREDATOR:

```
struct base_elem_seg0
{
  char name[30];
  int age;
  int height;
  struct base_elem_seg0 *next, *prev;
  struct base_elem_seg1 *seg1;
};
```

```
struct base_elem_seg1
{
  BYTE image[1000000];
  struct base_elem_seg1 *next, *prev;
};
```

3.5: Conceptual Limitation: Field Parameters

The example of the previous section also demonstrates the utility of NPTs with field parameters. The **`segment`** transformation requires a field name in order to know where to partition the original data type. Other examples of transformations with field parameters include ordered lists, B-trees, etc.

Because PREDATOR is a precompiler, it can easily perform type transformations that are much more complicated than just the addition of fields and operations, as provided by inheritance. We know of no other statically typed programming language (or data structure compiler) that supports the definition of complex type transformations.

3.6: Implementation Limitation: Code Efficiency

The efficiency of TPT code, particularly data structure code, is critical for most applications. Our work utilizes a precompiler optimizer to produce efficient code. There are several advantages to this approach:

1. When PREDATOR composes multiple NPTs, it performs compiler optimizations such as common subexpression elimination and partial evaluation.
2. PREDATOR expands inline code, thus removing runtime functional call overhead [12].
3. PREDATOR optimizes queries to determine the most efficient way to traverse a container [14, 3].

In the following sections, we report the results of three experiments using PREDATOR. We specifically examine the performance of generated code, the potential gains in software productivity, and the advantages of using plug-compatible data structures. It should be emphasized that these results are preliminary and should not be taken as definitive – merely indicative.

The results below are for programs of such simplicity that none of the advanced PREDATOR optimizations are used. The performance is based solely on the generated inline code. Further, all programs were compiled in C with the gcc compiler (version 1.40) with the same level of optimizations.

3.6.1: Experiment #1: Simple Arrays

The first experiment examines productivity gains and the potential efficiency of PREDATOR-generated code. We asked a group of eight professional programmers to write a simple four step program involving a trivial data structure – a static array in which records could be marked deleted. The four steps were:

- Eight records were copied from a static constant array into the target data structure. Each record was marked “not deleted.”
- An iteration was made over all records and each record was printed.
- Another iteration was made where each record that satisfied a supplied predicate was deleted.
- A final iteration printed all non-deleted records.

Each participant was asked to write three versions of the program: a quick and dirty version, a hand-optimized version, and a version using PREDATOR, and to note the time taken for each task. We executed the resulting programs 10,000 times using UNIX profiling tools to gather performance statistics. Table 2 summarizes the results.

Programming system	**Average time to write (min)**	**Average execution time (sec)**
Hand code (1st pass)	24.5 ± 7.5	17.5 ± 3.6
Hand code (optimized)	34.8 ± 9.1	12.2 ± 2.5
PREDATOR	8.0 ± 2.5	9.6 ± 0.1

Table 2: Simple array program results

These results of Table 2 suggest that (a) even in trivial programs, there are clear productivity benefits when programming with container abstractions, and (b) efficient code can be generated by a data structure compiler. We anticipate that these benefits will be magnified once more complicated data structures are used.

3.6.2: Experiment #2: Berkeley Quicksort

A second experiment involved writing a data-structure-generic quicksort algorithm and comparing its performance to BSD UNIX quicksort, a hand-optimized quicksort routine. In studying the BSD version, we noticed that:

- it only works on contiguous arrays,
- it is optimized for a data record size of 48 bytes, and
- it is quite difficult to understand. Modifying and debugging BSD quicksort is nontrivial.

Initially we intended to extract the data structure generic algorithm implemented by the BSD code, but discovered that it was too tightly coupled to the array data structure for this to be possible. Instead, we implemented the quicksort algorithm in [2] with a pivot selection and base case handling similar to the BSD version. This was important, as it assured the asymptotic complexity of both algorithms would be similar, thereby permitting a fair comparison to be made.

We exploited the data-structure-generic nature of the PREDATOR quicksort algorithm by plugging in two different container implementations. Again, this only required a trivial change in the container declaration; no change was needed in the PREDATOR quicksort algorithm.

The first data structure was an array, the data structure used by BSD quicksort. The second was a segmented record data structure: the primary segment simply contains a pointer to the secondary segment, which contains the data fields. This choice of segmentation is particularly appropriate because quicksort frequently swaps data records. In a segmented implementation, a record swap operation translates to a pointer swap operation whose time cost is independent of record size.

Table 3 compares the size of these three programs as computed by the UNIX word count (*wc*) utility. PREDATOR code is shorter and much easier to understand. Also note that the size of the generated PREDATOR code is much (30%-40%) larger than the precompiler source; PREDATOR is optimized for speed over size.

Source file	Number of words
BSD Quicksort	460
PREDATOR Source	323
Array (generated)	462
Segment (generated)	531

Table 3: Source code size comparison

The actual experiments involved sorting randomized sets of unique records. Randomization is important because the two algorithms use slightly different pivot selection methods. Uniqueness is significant because the PREDATOR [2] algorithm improves greatly when there exist duplicates and we did not want that to bias the results. Also, we ran each sample size many times to arrive at an accurate mean time.

The following graph shows the average sorting time for sample sizes ranging from 1000 to 110,000 records. The record size was set to 48 bytes. The sort key was composed of three fields, two 20 character strings and one integer with the primary field being a string.

Figure 5 shows a constant factor of time difference between the BSD execution time and that for each of the two PREDATOR examples. Thus, we conclude that the algorithms have the same asymptotic complexity. This means that we are measuring the true efficiency of the generated code, not the order of complexity of the different algorithm implementations.

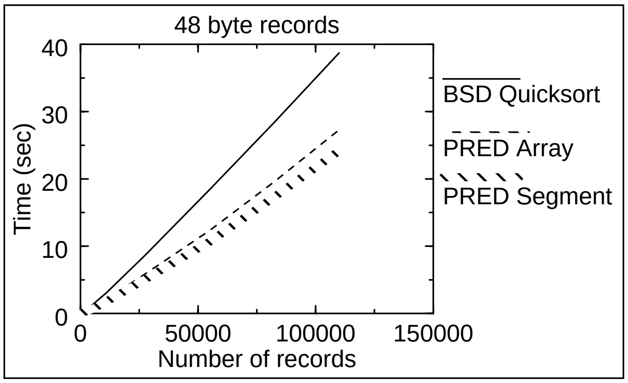


Figure 5: Quicksort performance

Figure 6 and Figure 7 show the effect of varying the size of the data record on the execution times. We measured this both for small and large data set samples and found similar behavior in both cases. Both the BSD quicksort and the PREDATOR array vary linearly in the size of the data record. This makes sense since they both copy the entire data record during a record swap. The PREDATOR segment case, however, is clearly superior to the other two because only the pointers to records, not the records themselves, are swapped. The very slight increase in the segment times is due to page faulting. Also note that there is a small region in which the BSD sort is superior to the PREDATOR array. This occurs at a record size of 8 bytes or below. This is due to the start-up costs associated with the segmentation and could be reduced even further with improvements to the PREDATOR optimizer.

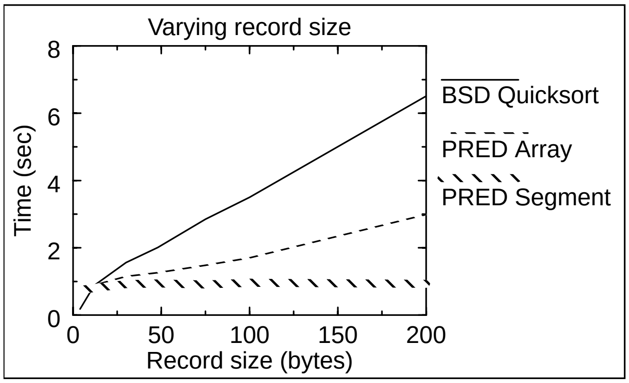


Figure 6: Quicksort of 1000 records

We believe the advantages that the PREDATOR quicksort demonstrates over the BSD quicksort are:

- The PREDATOR version is generic, i.e. it works for *any* unordered data structure.
- The PREDATOR version was quick to write, easy to understand and can be modified without much difficulty. We do not feel that this is true of the BSD quicksort.
- The performance of the prototype PREDATOR precompiler compares impressively against shipped, optimized code which is currently being used by many programmers.

3.6.3: Experiment #3: The rwho Utility

Many UNIX system utilities (*rwho*, *ls*, *df*, etc.) are fairly simple programs. In general, they involve data collection, placing the results in a data structure, sorting, and printing the results. These utilities are perfect candidates for re-implementation with PREDATOR.

rwho is a utility which prints a list of users who are logged on to the various machines in a local network. It reads this information from data files, stores data in an array, sorts the array, and prints the results. *rwho* uses BSD quicksort and stores the data in a static array. If there happen to be more than 100 entries, *rwho* fails. This is an example of unnecessary data structure dependencies that affect program behavior. PREDATOR-style programs with generic algorithms can easily remove this type of dependence.

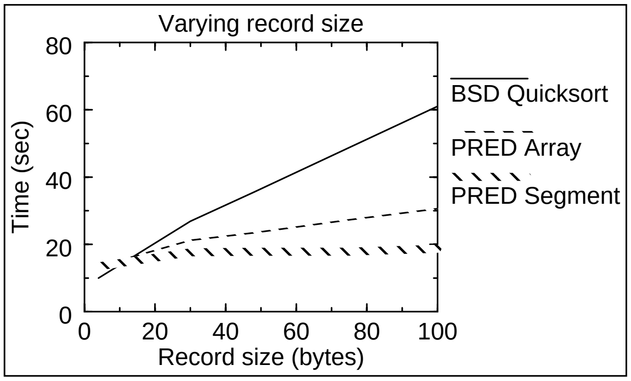


Figure 7: Quicksort of 99000 records

We rewrote *rwho* using PREDATOR in less than an hour. We chose a linked list data structure to remove the upper limit on the number of entries. While its performance is (marginally) better than that of the original *rwho*, the important point here is that PREDATOR helped us improve the functionality of the *rwho* program in a convenient manner.

4: Assessment and Conclusions

We believe that data structure compilers show promise in offering programming productivity gains without sacrificing performance. We have identified conceptual, design, and implementation limitations in traditional parameterized types (TPTs). We believe that TPTs alone are not sufficiently powerful to form the foundation of a data structure compiler.

To overcome the limitations of TPTs, we have presented several techniques for capturing data structures as NPTs (i.e. type transformations). Using PREDATOR, NPTs can be combined to produce efficient implementations of complex data structures. In addition, by designing NPTs to have the same interface, plug-compatibility and interchangeability is assured. This *significantly* simplifies the task of building a data structure compiler and enhances opportunities for software reuse.

PREDATOR is based on concepts from databases, compilers, transformation systems, and domain modelling. By themselves, the ideas presented here are not new. However, we do believe their combination is unique for solving important problems in data structure compilers.

Our preliminary results using PREDATOR are promising. We have found that programmers are able to write data-structure-independent code more easily, that the generated code is efficient, and that software reuse is accomplished. Moreover, our model is sufficiently robust to handle NPTs for a wide variety of data structures. Finally, the PREDATOR model does not seem to be incompatible with related concepts such as concurrency, persistence, and data distribution. Augmenting the model for these features is future work.

5: References

[1] ACM. Next Generation Database Systems. *Communications of the ACM*, October 1991.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.

[3] D. S. Batory. Concepts for a DBMS Synthesizer. *Symposium on Princ. of Database Sys.*, 1988.

[4] D. S. Batory. The Genesis Database System Compiler: User Manual, University of Texas TR-90-27.

[5] T. Biggerstaff and A. Perlis. *Software Reusability*, ACM Press, 1989.

[6] G. Booch. *Software Components with ADA*, Benjamin/ Cummings Publishing, 1987.

[7] G. Booch and M. Vilot. The Design of the C++ Booch Components. *OOPSLA ECOOP 90*, ACM Press, 1990.

[8] C. Chambers, D. Unger, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. *OOPSLA 89*, ACM Press, 1989.

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

[10] D. Cohen. *AP5 Manual*, USC Information Sciences Institute, 1991.

[11] C. J. Date. *An Introduction to Database Systems*, Addison-Wesley, 1983.

[12] J. Davidson. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Trans. on Soft. Engr.*, February 1992.

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

[14] H.F. Korth and A. Silberschatz. *Database System Concepts*, McGraw-Hill, 1991.

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

[16] D. Lea. libg++, The GNU C++ Library. *C++ Conference USENIX Association*, Denver, CO 1988.

[17] D. G. McNicoll, C. Palmer, et al. *Common Ada Missle Packages (CAMP) Volume I: Overview and Commonality Study Results*, AFATL-TR-85-93, May 1986.

[18] D. G. McNicoll, C. Palmer, et al. *Common Ada Missile Packages (CAMP) Volume II: Software Parts Composition Study Results*, AFATL-TR-85-93, May 1986.

[19] G. Novak. Software Reuse by Compilation through View Clusters. Submitted for publication in *IEEE Trans. on Soft. Engr.*, 1992.

[20] 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.

[21] R. Prieto-Diaz and G. Arango. *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[22] J. Schmidt. Some High Level Language Constructs for Data of Type Relation, *ACM Trans. on Database Sys.*, 1977.

[23] R. Snodgrass. *The Interface Description Language*, Computer Science Press, 1989.

[24] B. Stroustrup. *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1991.

[25] D. Volpano and R. Kieburtz. Software Templates, *Proc. Eighth Intl. Conf. on Soft. Eng.*, IEEE Computer Society, 1985.

[26] D. Weiss. *Synthesis Operational Scenarios*, Software Productivity Consortium, Inc, 1990.