Compiler-Enforced Memory Semantics in the SACLIB Computer Algebra Library

David G. Richardson

Technical Report DU-CS-05-14
Department of Computer Science
Drexel University
Philadelphia, PA 19104
December, 2005

Compiler-Enforced Memory Semantics in the SACLIB Computer Algebra Library

A Thesis

Submitted to the Faculty

of

Drexel University

by

David G. Richardson

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Science

December 2005

© Copyright December 2005
David G. Richardson.

Table of Contents

List of Figures iii

Abstract iv

1. Introduction 1

2. Related Work 4

   2.1 Garbage Collection 4

   2.2 Static Analysis 5

3. Template Metaprogramming 7

4. Existing SACLIB practice 9

   4.1 The source-sink idiom and its limitations 9

      4.1.1 Failure to release resources 9

      4.1.2 Uninitialized Delete and Double Delete 9

      4.1.3 Source-sink functions difficult to maintain 11

   4.2 Stack-before-heap idiom 11

   4.3 Memory Passing and Return Value Conventions 12

5. Improvements to SACLIB 14

   5.1 Replacing the source-sink idiom with RAII 14

   5.2 Encapsulation of the stack-before-heap idiom 16

   5.3 Recursively Fixed Iterator 19

      5.3.1 rec_fixed_itr implementation 24

   5.4 simple_ptr 27

   5.5 Proving the Memory System Leak Free and Double Delete Free 29

6. Performance Testing 32

7. Writing Code for the Optimizer 34

8. Automatic Conversion 36

9. Conclusion 38

Bibliography 39

List of Figures

4.1.1 Incorrect use of the source-sink idiom in the SACLIB routine MAIPDME results in a memory leak. 10

5.1.1 The heap `array` class cannot leak memory. 15

5.1.2 The new version of the SACLIB routine MAIPDME is leak-free—even without the call to FREEARRAY that was missing in Figure 4.1.1. 16

5.2.1 The stack `array` class efficiently stores elements on the system stack. 17

5.2.2 Memory leaks are automatically avoided and the system stack is used for storage using the stack `array` class. 18

5.3.1 P is a pointer to a SACLIB matrix. The gray memory cells contain pointers and are struct(P). The white memory cells contain integers and are cont(P). 22

5.3.2 The metafunction Dim computes the dimensionality of a type. 26

5.3.3 Outline of `rec_fix_itr` implementation technique 28

Abstract

Compiler-Enforced Memory Semantics in the SACLIB Computer Algebra Library

David G. Richardson

Advisor: Werner Krandick

We present a memory management subsystem for the computer algebra library SACLIB that removes the potential for memory leaks or double deletes in applications using the system. The system encapsulates the management of arrays that are allocated on the heap or on the system stack. The system makes arrays responsible for their own memory management, and enables the compiler to prevent other parts of SACLIB from managing array memory. To prove that our memory module and all applications using it are leak free and double delete free we introduce a new iterator concept and implement a model of that concept using generic programming techniques such as template metaprogramming. Our innovations reduce the amount of code responsible for array memory management from $10,000$ lines of code to $2,000$ lines of code. Using hardware performance counters we show optimizing compilers are capable of avoiding any runtime overhead.

1. Introduction

Memory management is a critical component of any computer algebra library. Correct memory management is required to ensure the correctness of subroutines; efficient memory management facilitates improved execution speed and the processing of larger data sets.

The SACLIB [11, 24] library of computer algebra programs is used as the basis of the quantifier elimination systems QEPCAD [12, 22, 23] and QEPCAD B [8, 9]. However, SACLIB contains a number of memory management defects, all of which occur outside of SACLIB's garbage collector. Memory leaks involving dynamically allocated arrays have caused problems in certain large computations [32]. While runtime-tools such as Valgrind [52], Electric Fence [16], and Purify [25] can be used to detect some memory defects, the tools cannot guarantee their absence. This is unfortunate because dynamic arrays are used extensively in a weakly typed computer algebra system such as SACLIB.

We port SACLIB from C to C++ and introduce a memory management subsystem that removes the potential for any memory leaks or double deletes in applications using the system without requiring any runtime overhead. The system encapsulates the management of arrays that are allocated on the heap or on the system stack. The system makes arrays responsible for their own memory management and enables the compiler to prevent other parts of SACLIB from managing array memory. To prove that our memory module and all applications using it are leak free and double delete free we introduce a new iterator concept (see Section 5.3) and implement a model of that concept using generic programming techniques such as template metaprogramming. Our innovations reduce the amount of code responsible for array memory management from 10,000 lines of code to 2,000 lines of code.

Our memory management system consists of an iterator to control memory usage and of array classes to handle memory allocation. Using template metaprogramming [2, 1, 53] we implement a model of our iterator concept that handles arrays of arbitrary dimension. Our metaprogramming techniques are similar to those used by Musser and Schupp in the SuchThat [36, 37, 45] computer algebra library and in library assisted code optimizations [20, 44]; Schreiner [43] details three other

computer algebra libraries that use generic programming. However, our implementation is the first to provide iterators capable of protecting memory of arbitrary dimension.

The array component of our memory management system (Section 5.1 and 5.2) is similar to the `std::vector` class found in C++'s Standard Template Library [29, 49, 30], but our implementation is faster than the `std::vector` class of the implementations shipped with our compilers, see Table 6.1.

In addition, our memory management subsystem offers numerous software maintenance advantages. It allows some runtime bounds checking to be performed during the development process without the need for external tools. The system provides a well-defined architecture for future extensions of the non-garbage collected memory subsystem. By embedding memory semantics directly in the C++ type system, the assumptions functions make about memory management are clearly announced and enforced by their prototypes.

Our memory management subsystem provides the same service as garbage collection: memory management is automatically and correctly performed regardless of the discipline employed by programmers. However, the runtime [38] overhead of garbage collection makes it unsuitable for performance critical applications. Additionally, the unpredictable pauses caused by garbage collection makes it unsuitable for hard real time systems [5]. As our method requires no unpredictable pauses or runtime overhead, it is suitable for a wider range of applications than garbage collection.

To attain these benefits, it was necessary to port SACLIB from C to C++. We implemented code transformation scripts to convert most of the code from K&R C [31] to ANSI C++ [29, 49]; the remainder of the code was related to floating point exception handling and was converted by hand.

In Section 2 we will discuss related methods. In Section 3 we will provide a brief tutorial on the implementation technqiues required for our method. In Section 4 we detail the limitations of SACLIB's design that resulted in a strong potential for memory leaks. In Section 5 we present our memory management subsystem, outline the general implementation techniques employed, and give examples of its use. In Section 5.5 we prove that our memory management subsystem and the applications using it are leak and double delete free. In Section 6 we report the results of performance experiments on our memory management subsystem. Through the use of hardware

performance counters and the PAPI library [26] we are able to report timings with a resolution under 50 nanoseconds. In Section 7 we explain why our implementation can achieve no runtime overhead. In Section 8 we explain how we ported SACLIB from C to C++. We conclude in Section 9 with a discussion of the effectiveness of our method and lessons learned from the design and implementation of our subsystem.

2. Related Work

A program is memory safe if all of its memory management and access are performed correctly. The ability to provide programming languages and tools to allow programs to be automatically memory safe has been a topic of much research.

With respect to memory management, the two main research results have been garbage collection and static verification via stronger typing.

2.1 Garbage Collection

Garbage collection has been a highly pursued strategy in helping programmers automatically achieve correct memory management. An ISI Web of Science [46] search for "garbage collection" from 1990-2005 returns 512 articles. The Open Directory Project [41] catalogs 90 programming langauges as being garbage collected langauges. Representaive examples of such garbage colelcted languages are Java [18], Perl [55], Python [42], Ruby [33], and Lisp [19, 3].

These languages generally provide bounds checking in an attempt to prevent invalid memory access (for example, Java) or provide transparent dynamic memory management that ensures that all memory access are to allocated memory (for example, Perl).

While these techniques can allow for tremendous gains in programmer productivity compared to languages with manual memory management, the price for runtime efficiency can be quite high. Necula [38] has conducted a series of benchmarks where C programs were converted to use state of the art garbage collection and bound checking algorithms. For some work loads, the overhead introduced was almost a factor of 10 degradation of runtime performance. The vast majority of the slow down was due to the garbage collector.

In the domain of performance critical and real-time applications, the variability of garbage collections [5] and its potentially high runtime overhead [38] prevent it from being a viable technique.

2.2 Static Analysis

The CCured [39, 13, 21, 38] system consists of a set of language extensions to annotate pointer usage of C programs and a CCured to C source to source translator. The goal of CCured is to use stronger typing to statically verify the safety of memory operations. When the memory safety of an operation cannot be verified statically, CCured inserts code to perform runtime verification of memory safety.

Correct memory management is obtained by altering `malloc` to return memory that is manged by a garbage collector. Calls to `free` are set to do nothing.

The foundation of the CCured type system are the following pointer qualifiers: `SAFE`, `SEQ`, `WILD`. Note: CCured has additional refinements of these type qualifiers that will not be discussed here. The type qualifiers have the following meanings

- `SAFE` - The pointer is only ever used for memory dereference. There is no pointer arithmetic performed or type casts preformed on the pointer.
- `SEQ` - The pointer is involved in pointer arithmetic. Type casts are still not performed on the pointer.
- `WILD` - Due to type casts, the static type of the pointer may not provide an accurate description of the proper semantics to be ascribed to the memory it points to.

`SAFE` pointers must be checked at runtime to ensure they are not null before dereferences. `SEQ` pointers must have a null check and a bounds check performed at runtime. `WILD` pointers must have a null check, bounds check, and a semantic check of the memory they refer to at runtime.

CCured must store additional data about memory to allow such checks to be performed. The library uses two approaches:

1. pointers only used in CCured applications have their metadata stored with the pointer. This results in CCured pointers having a different size then built in pointers. This has a performance advantage in that the metadata is stored with the pointer that needs it.

2. For pointers that must be used with external libraries that do not use the CCured types, the metadata is stored separately from the pointer. This representation of CCured pointers can be passed to external functions that expect a one word pointer.

In order to limit the programmer involvement as much as possible, CCured has a type inference algorithm that tries to deduces the correct pointer qualifiers to use. Obviously, the runtime overhead introduced by CCured is minized when it can find the least expensive type qualifer that will garantee memory saftey for a pointer.

However the CCured system is not fully automatic. Neccula [38] estimates that converting a 100 kloc program to use the CCured type system will require approximately 30 hours. Programs using a large number of C "dirty tricks" are expected to take longer. The steps to convert a C program to use CCured are as follows:

1. Alter the projects build process to use the CCured translator.

2. Annotate program constructs CCured cannot process (for example `sizeof` and variable argument functions).

3. Review the type casts CCured cannot prove the safety of and manually annotate the safe casts.

4. Resolve linker errors with external libraries caused by incompatibilities between some CCured pointer types and native pointers.

5. Test and debug the the CCured version of the applications.

Due to the metadata required for the CCured type inference algorithm to work, external libraries can be problematic. In the worst case, a programmer must make hand wrapper functions for all external programs that are called from a CCured application. The CCured system provides such wrappers for the C standard library.

3. Template Metaprogramming

The implementation of the methods presented here require a great deal of template metaprogramming. For readers not familiar with template metaprogramming, this section will provide a brief introduction. For a more comprehensive treatment, readers are referred to the following references: [1, 2, 53].

The template instantiation mechanism was discovered to be a Turing complete functional language by Erwin Unruth [53]. Template metaprograms are evaluated at compile time. The evaluation mechanism supports the computation of either type or integral constants.

As an example of a type computation consider the following metaprogram to compute a pointer to a given type:

```
1: template <typename T>
2: struct pointerTo;
3:
4: template <typename T>
5: struct pointerTo{
6:   typedef T* type;
7: };
```

Lines 1-2 are a template class declaration. For a template metaprogram, this serves the same purpose a prototype serves for a function. Lines 4-6 are a template class definition. For a template metafunction this serves the same purpose as a funciton definition does for a function. Line 6 use a typedef to provide the result of the type computation as the nest typedef type. This is a standard convention used in template metaprograms [1, 6]. The template parameters of a template metafunction serve as its arguments.

The `pointerTo` metaprogram would be used as follows:

```
poinerTo<int>::type p; //same as int* p;
```

In this example, it is clearly not advantageous to use the `pointerTo` metafunction when `int*` could just as easily have been written. However, when used inside another template metafunction, the argument to `pointerTo` may be another template parameter. In Section 5.3.1 we show how we employed template metaprogramming techniques to create useful functionality.

As an example of computing an integral constant, consider the computation of if a type is a pointer:

```
 1: template <typename T>
 2: struct isPointer;
 3:
 4: template <typename T>
 5: struct isPointer{
 6:    enum {value=false};
 7: };
 8:
 9: template <typename T>
10: struct isPointer<T*> {
11:    enum {value=true};
12: };
```

Lines lines 1-2 are the metafunction prototype. Lines 4-7 are a metafunction definition for the general case when T is any type. Lines 9-11 are a partial template specialization for the case when T is pointer. The use of such template specialization provides the basis for all control structures used in template metaprograms. The enums are used to store the result of the computation, and are accessed in a similar way as the nested type is for `pointerTo`.

Iteration is accomplished by recursive instantiation of templates.

4. Existing SACLIB practice

4.1 The source-sink idiom and its limitations

SACLIB uses the source-sink idiom [50] for all memory management outside of its garbage collector and some performance critical sections. In the source-sink idiom a *source function* is used to request a handle to a system managed resource. The handle allows clients of the source function to access the resource without any knowledge of how it was acquired by the source function. When the handle provided by the source function is no longer needed, it is passed to a *sink function* which releases the resource referred to by the handle. A resource leak occurs when all handles to a resource have been lost and, as a result, the resource cannot be released by a sink function. Examples of source-sink pairs are the library functions `malloc`/`free` and SACLIB functions such as `GETARRAY`/`FREEARRAY` and `GETMATRIX`/`FREEMATRIX`.The source-sink idiom gives rise to two types of programming mistakes, and it makes it hard to change the implementation of the source-sink pair.

4.1.1 Failure to release resources

The source-sink idiom cannot guarantee that all resources acquired from a source function will be released by a sink function. Indeed, memory leaks in SACLIB subroutines prevented an experiment involving a large number of polynomials from successfully running to completion [32]; Figure 4.1.1 shows an example of a SACLIB routine with a memory leak.

4.1.2 Uninitialized Delete and Double Delete

The source-sink idiom cannot guarantee that the source and sink functions are called for valid handles. When the same handle is passed to a sink function more than once it is called *double deletion*. When a handle not initialized by a source function is passed to a sink function this is called *uninitialized deletion*.

The compiler does not prevent SACLIB from calling sink functions on uninitialized handles

```
Step3: /* Compute a list of the order 2 minors. *}

    C = GETARRAY(2);

        C[0] = 0;
        C[1] = 1;
        L = NIL;
        while (C[0] >= 0) {
           i = C[0];
           j = C[1];
          B1 = IPPROD(r,A[n-2][i],A[n-1][j]);
          B2 = IPPROD(r,A[n-2][j],A[n-1][i]);
           B = IPDIF(r,B1,B2);
           c = CSFAM(2,C);
           L = COMP2(B,c,L);
           LEXNEXTC(n,2,C,&t);
        }
        L = INV(L);

    /* FREEARRAY(C) missing. {*}/
```

Figure 4.1.1: Incorrect use of the source-sink idiom in the SACLIB routine MAIPDME results in a memory leak.

or handles that have already been passed to a sink function. But such calls produce undefined runtime behavior.

4.1.3 Source-sink functions difficult to maintain

SACLIB uses pointers for all of its handles. Because the compiler has no control over what functions are called on these pointers, clients of the source-sink pairs must check the implementation of the source-sink pairs to determine what operations may safely be applied to the handles. Not surprisingly, this poor information hiding has all of the problems one would expect [40]. In addition to making current development more difficult by requiring the programmer to be aware of the implementation of the source-sink pairs, changes to the source-sink pairs can cause other previously working parts of SACLIB to become defective.

4.2 Stack-before-heap idiom

For some performance critical sections of SACLIB the stack-before-heap idiom is used for memory management. The idiom consists of allocating memory on the stack that is large enough for the majority of inputs that SACLIB is expected to run on. In the event that the amount of stack memory is sufficient for a given input, the overhead of both allocating heap memory and heap fragmentation are avoided. This technique is similar to the small string optimization [51, 35], although SACLIB does not use unions. The stack-before-heap idiom is more error prone than the source-sink idiom. It is more complicated and does not lend itself to being encapsulated in functions. As a result, all uses of the stack-before-heap idiom require the programmer to manually implement all parts of the idiom. Even when implemented correctly, this detracts from the readability of the code. Even when the stack-before-heap idiom is implemented correctly, it is not always a performance optimization. It is only a performance optimization when the amount of stack memory is sufficient for most inputs. If inputs frequently (or even worse, always) require more memory than is allocated on the stack, the stack-before-heap idiom offers the opportunity for reduced performance. Furthermore, because the stack-before-heap idiom is manually implemented, each occurrence must be manually checked to see if the idiom is in fact an optimization. While it is

likely that the stack-before-heap idiom originally was an optimization for a meaningful input set, it is unlikely that the use of the idiom will be rechecked as system maintenance is performed and the program is used with new input sets. There is no way to automatically detect where all uses of the stack-before-heap idiom have been used. In even a moderately sized system such as SACLIB, it becomes infeasible to test each occurrence of the stack-before-heap idiom as the system evolves.

4.3 Memory Passing and Return Value Conventions

All memory in SACLIB is referred to by pointers. Functions requiring the ability to be called polymorphically with respect to the value type of a pointer are implemented using void pointers. Neither pointers or the data they point to are declared `const`.

SACLIB currently uses pointers to return values from functions that return more than one value. By convention, the output parameters of a function are placed at the end of the argument list and the argument name have a trailing underscore appended argument name.

Pointers are one of the most powerful, flexible—and error prone—features of the C and C++ programming language. Because of their flexibility, it is generally the case that a function taking pointers as its arguments does not require all of the functionality offered by pointers. This results in function prototypes that fail to convey as much information about the function as more strongly typed prototype can. This lack of information results in three types of maintenance problems.

1. A function taking raw pointers does not provide any information about its expectations regarding memory ownership. Clients of the function are forced to read the functions specification and implementation in ordered to determine if the function frees or allocates memory passed via pointers. This problem is exacerbated as the dimensionality of the pointers increases. Indeed, one of the programming construct discovered in SACLIB that was likely to result in memory leaks occurred when functions tried to dynamically resize elements of multidimensional arrays passed to them by clients.

2. Pointers intended to be used solely as return values are capable of being used for pointer arithmetic. As it is likely the pointers passed in for return values will be obtained by taking

the address of automatic variables, any accidental pointer arithmetic is likely to result in undesirable side effects like memory corruption and segmentation faults.

3. Functions may have defects added during maintenance and the function prototype cannot convey any sense of correctness. This makes debugging more difficult because memory problem require the investigation of all functions on the execution path resulting in the memory defect.

5. Improvements to SACLIB

5.1 Replacing the source-sink idiom with RAII

A well know replacement for the source-sink idiom is the Resource-Acquisition-Is-Initialization (RAII) idiom [2, 34, 50, 48, 47]. This idiom is used extensively in modern C++ libraries, for example containers in the Standard Template Library [4], the Loki smart pointer classes[2], and the Boost smart pointer library [6].

In the RAII idiom, responsibility for resource acquisition is delegated to a class specifically designed to safely acquire and release resources. The class is responsible for acquiring resources in its constructor and freeing all resources it has acquired in its destructor. The C++ standard [29, 49] guarantees that an object will have its destructor called when it goes out of scope.

The RAII idiom drastically reduces the severity of errors the programmer can make. Since the programmer does not need to remember to release resources, it is much simpler to write multiple exit point functions. The only mistakes that can be made involve the amount of time the resources are held. By acquiring resources in an overly broad scope it is possible for the program to be done with the resources before they go out of scope. Even though resources may be held too long, it is not possible to waste an unbounded amount of a resource through leaks.

SACLIB's GETARRAY/FREEARRAY source-sink pair was replaced with an `array` class. To show how the RAII idiom can be implemented we give the following minimal implementation, see Figure 5.1.1, of the `array` class. The actual implementation contains additional features such as the performance enhancements detailed in Section 5.2, the typedefs and methods required to make the array model the safe parts of the STL Random Access Container conceptn, and the ability to resize the array after the the constructor is called. Figure 5.1.2 shows that the use of our array class automatically removes memory leaks from SACLIB.

Remark 5.1.1. An `array2d` class is also provided for the new SACLIB. It provides RAII memory management for two dimensional arrays and offers a similar interface to the `array` class.

```
template<typename T>
class array{

   public:

      //ctors acquire memory
          array()        : activeArray_(NULL)     {};
          array(size_t n): activeArray_(new T[n]) {};

      //dtor releases memory
          ~array(){ delete[] activeArray_; };

      //allow the array class to be used like a built in array
          T& operator[](ptrdiff_t index){
             return activeArray_[index];
          };//operator[]

         const T& operator[](ptrdiff_t index) const {
            return activeArray_[index];
         };//operator[] const

   private:

       T *activeArray_;

};//array
```

Figure 5.1.1: The heap `array` class cannot leak memory.

```
Step3: /* Compute a list of the order 2 minors. */

        array<BDigit> C(2);
        C[0] = 0;
        C[1] = 1;
        L = NIL;
        while (C[0] >= 0) {
            i = C[0];
            j = C[1];
           B1 = IPPROD(r,A[n-2][i],A[n-1][j]);
           B2 = IPPROD(r,A[n-2][j],A[n-1][i]);
            B = IPDIF(r,B1,B2);
            c = CSFAM(2,C.begin());
            L = COMP2(B,c,L);
           LEXNEXTC(n,2,C.begin(),&t);
        }
        L = INV(L);
```

Figure 5.1.2: The new version of the SACLIB routine MAIPDME is leak-free—even without the call to FREEARRAY that was missing in Figure 4.1.1.

5.2 Encapsulation of the stack-before-heap idiom

The principle draw back to the stack-before-heap idiom is the inability to encapsulate the idiom in C functions. In addition to removing the need for the source-sink idiom, the `array` class also provides a convenient place to encapsulate the stack-before-heap idiom. Figure 5.2.1 is an example of an implementation of the array class that provides an encapsulated version of the stack-before-heap idiom. Note that the implementation actually used in SACLIB contains additional features such as the typedefs and methods required to make the array model the safe parts of the STL Random Access Container concept and the ability to resize the array after the the constructor is called. Partial specialization is also used to ensure there is no overhead when `N` is zero. Figure 5.2.2 shows an `array` used to provide a safe alternative to the stack-before-heap idiom.

The `array` class has been given an additional template parameter `N` to specify the number of elements to be stored on the stack and the private member array `stackStorage_` in which to store up to `N` elements of type `T`. In the event that a user instantiates an array class requiring less than `N` elements, the array constructor will set the `activeArray_` member to `stackStorage_` and there

```
template<typename T, size_t N>
class array{

   public:

      //ctors acquire memory
         array()         : activeArray_(stackStorage)                        {};
         array(size\_t n): activeArray_( n <= N ? stackStorage : new T[n] ) {};

      //dtor releases memory if needed
         ~array(){
            if(activeArray != stackStorage) delete[] activeArray_;
         }//~array

      //allow the array class to be used like a built in array
         T& operator[](ptrdiff_t index){
            return activeArray_[index];
         };//operator[]

         const T& operator[](ptrdiff_t index) const {
            return activeArray_[index];
         };//operator[] const

   private:

      T *activeArray_;
      T stackStorage_[N];

};//array
```

Figure 5.2.1: The stack `array` class efficiently stores elements on the system stack.

```
void encapsulated_stack_before_heap_example(){

   //how much stack memory to use
      const size_t stack_size(50);

   //"compute" the size of array
      int n;
      cout << "What size array do you want?";
      cin  >> n;

   //acquire resources with a constructor using
   //the stack-before-heap idiom
      array<int,stack_size> a(n); //pass desired array size to
                                  //constructor

}//encapsulated_stack_before_heap_example
```

Figure 5.2.2: Memory leaks are automatically avoided and the system stack is used for storage using the stack `array` class.

will be no need to allocate memory from the heap. If a user instantiates an array that requires more than `N` elements, it will set `activeArray_` to a pointer obtained by calling new. This will incur the full overhead of using the heap, and the memory in `stackStorage_` will go unused. This is the same situation that occurs when the stack-before-heap idiom is implemented manually. However, because the array based version combines the stack-before-heap idiom with the RAII idiom the potential for a memory leak is lost.

While the removal of memory leaks is the most substantial benefit obtained of this implementation, there is a another benefit. In the manual implementation of the stack-before-heap idiom it is difficult to verify that the amount of storage allocated on the stack is sufficient to be a performance optimization because there is no central location to check. With the encapsulation of the idiom in the array class a check can be placed in the array constructor. With conditional compilation all overhead of the check can be avoided in production code. By altering only methods of the array class, all clients of the array class can be run on representative inputs to see if the proper amount of stack storage space is being allocated.

5.3 Recursively Fixed Iterator

The heavy use of the source-sink idiom in SACLIB offered the opportunity to allow functions to clearly express and enforce memory ownership expectations through their prototypes. The key insight required to allow memory ownership to be expressed by the function prototype is this: non-sink functions taking pointers to memory manged by the source-sink idiom should not deallocate the memory referred to by the pointers.

In the development of the theoretical framework used to describe our memory management system and theorems about various aspects of its correctness we will use the following definitions from the STL [4]:

A concept [4, 36, 37] is a collection of constraints on a type. Constraints may deal with the syntax and semantics of a type or the time/space complexity of operations supported by the type. The following definitions from the C++ Standard Template Library (STL) [4, 29, 49] regarding concepts will be used.

Definition 5.3.1. (from STL) A C++ type X is a *model* of a concept when it meets all the constraints of the concept. Note that a concept specifies the minimal set of constraints a type must have to be a model of the concept. Therefore even when a type has more features than those described by a concept it is still considered a model of the concept.

Definition 5.3.2. (from STL) A concept C2 is a *refinement* of the concept C1 if and only if C2 specifies all the constraints of C1 plus additional constraints not specified by C1. When C2 is a refinement of C1 we say that C2 is *stronger* concept than C1 and C1 is *weaker* concept than C2.

We use the following concept from the STL:

Definition 5.3.3. (from STL) A type X is a model of the *Default Constructible* concept if and only if it has a default constructor.

Remark 5.3.4. A default constructor is defined by the C++ standard [29, 49] as a constructor that may be called without any user supplied arguments.

Remark 5.3.5. All built-in C++ types and C-style structs of built-in C++ types are models of the Default Constructible concept.

We introduce the following definitions to facilitate the discussion of the ways in which SACLIB functions used pointers passed in by their clients:

Definition 5.3.6. Let T be a C++ type. If T is a user defined type T::operator*, T::operator->, and T::operator[] have the usual C++ meaning of referring to T's member functions operator*, operator->, and operator[]. If T is a built in pointer type T::operator*, T::operator->, and T::operator[] then they refer to the functionality offered by the compiler when *, ->, and [] are applied to instances of type T.

Definition 5.3.7. A C++ type T is **pointer-like** if T::operator* and T::operator->are valid function calls with pointer semantics.

Definition 5.3.8. A C++ type T is **fully pointer-like** if T::operator*, T::operator->, and T::operator[] are valid function calls with pointer semantics and T supports pointer arithmetic.

Definition 5.3.9. Let F be a C++ function. If F never uses `const_cast` to remove `const` from a cv-qualified type, F is a *const-respecting function.*

Remark 5.3.10. The old SACLIB does not contain any `const` qualified variables or `const_cast`s. Therefore all functions in the old SACLIB are const respecting. The new SACLIB contains `const` qualified variables, but no `const_casts`.

Definition 5.3.11. Let F be a C++ function. Let the return type of F be denoted by $\mathrm{ret}(F)$. Let T be a type in the C++ type system. Let the **dimension** of the type T be denoted $\dim(T)$ and the **dimension** of all instances t of type T be denoted $\dim(t)$. If T is not pointer like, then $\dim(T) = 0$. If T is pointer like, then $\dim(T) = 1 + \dim(\mathrm{ret}(T\text{::operator*}))$. For all t, $\dim(t) = \dim(T)$.

Definition 5.3.12. Let T be a C++ type, and let t be an instance of T. Then the *index span* of t is defined as

$$\text{span}(t) = \begin{cases} \emptyset & T \text{ not fully pointer like;} \\ \{n \in Z \mid *(t+n) & \text{otherwise.} \\ \quad \text{is a valid memory dereference}\} & \end{cases}$$

Definition 5.3.13. Let T be a C++ type, and let t be an instance of T. Then the *directly dereference-able elements* of t is defined as

$$\text{DD}(t) = \begin{cases} \emptyset & T \text{ is not fully pointer like;} \\ \{*(t+n \in Z) \mid n \in \text{span}(t)\} & \text{otherwise.} \end{cases}$$

Definition 5.3.14. Let T be a C++ type, and let t be an instance of T. Then the *dereference-able closure* of t is defined as

$$\text{DC}(t) = \begin{cases} \emptyset & \text{if } \dim(T) = 0; \\ \text{DD}(t) \cup_{i \in \text{DD}(t)} \text{DC}(i) & \text{otherwise.} \end{cases}$$

Definition 5.3.15. Let T be a C++ type and let t be an instance of T. Then the *structure* of t is defined as

$$\text{struct}(t) = \{i \in \text{DC}(t) \mid \dim(i) > 0\}.$$

Definition 5.3.16. Let T be a C++ type and let t be an instance of T. Then the *contents* of T are defined as

$$\text{cont}(t) = \text{DC}(t) - \text{struct}(t).$$

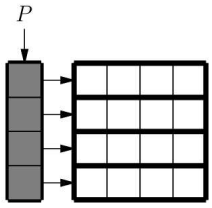

Figure 5.3.1: P is a pointer to a SACLIB matrix. The gray memory cells contain pointers and are struct(P). The white memory cells contain integers and are cont(P).

Using these definitions, we may classify C++ functions in the following three ways:

Definition 5.3.17. Let F be a C++ function. Let F take an argument p of type P. Let P be fully pointer like. Then, F is a **memory manager** of p if and only if F writes to struct(p) or deallocates memory referred to by any of the elements in struct(p) in at least one of its execution paths.

Definition 5.3.18. Let F be a C++ function. Let F take an argument p of type P. Let P be fully pointer like. Then, F is a **contents user** of p if and only if F reads or writes any elements of cont(p) in at least one of its execution paths.

Definition 5.3.19. Let F be a C++ function. Let F take an argument p of type P. Let P be fully pointer like. Then, F is a **contents only user** of p if and only if F is a contents user of p and F is not a memory manger of p.

Theorem 5.3.20. *Let F be a C++ function. Let F be a contents only user of all fully pointer like arguments it is passed. Then F does not leak memory or double delete memory referred to by its fully pointer like arguments.*

Proof. F does not leak memory because as a contents only user of its fully pointer like arguments it cannot write to their structure. Because the structure is not written to none of the existing

references to allocated memory can be lost and therefore there cannot be a memory leak. F cannot cause a double deletion because as a contents only user of its fully pointer like arguments it will never delete any of the memory referred to by the structure of its fully pointer like arguments. □

Theorem 5.3.20 offers a condition for verifying that a function does not leak or double delete any memory referred to by its fully pointer like arguments. However, without a tool to identify which functions are memory managers and which functions are contents only users developers must manually determine if the memory correctness of each function they are debugging for memory leaks or double deletes. This is a tedious process because when memory errors occur they are generally not confined to a single function. Furthermore, maintenance programming results in the need for maintenance programmers to constantly on guard against introducing memory leaks and double deletes.

Motivated by the desire to bring the ability to declare a function to be a contents only user into the C++ type system we define the following concept:

Definition 5.3.21. Let T be a C++ type. T is a model of the *recursively fixed iterator* concept if

1. T provides T::operator*, T::operator->, and T::operator[] with pointer semantics,
2. T provides all pointer arithmetic operators that are not self-modifying (+,-, <, <=, >, >=, ==), and
3. For all instances t of type T const-correct functions may write to cont(t), but if they attempt to write to struct(t) they will not compile.

Using operator overloading and template metaprogamming we have implemented `rec_fixed_itr` as a model of the recursively fixed iterator concept.

With the availability of the `rec_fixed_itr` class functions can become both more expressive and capable of protecting memory structure passed to them. Consider:

```
void might_be_a_memory_manger_of_p(int*** p);
void contents_only_user_of_p      (rec_fixed_itr<int***> p);
```

From the function prototype, neither the compiler nor maintenance programmers have any knowledge of the usage patterns the developers of `might_be_a_memory_manager_of_p` intended. However, the function prototype of `contents_only_user_of_p` clearly states it is a contents only user via the argument type `rec_fixed_itr<int***>`. If the definition of `contents_only_user_of_p` ever attempts to be a memory manager of p, the compiler will prevent compilation of `contents_only_user_of_p`. Furthermore, we implemented `rec_fixed_itr` such that the compiler will emit a diagnostic error that indicates the exact line of `contents_only_user_of_p`'s definition that attempts to manage memory. Schreiner's criticism regarding template parameter validation [43] no longer applies; indeed, the static assert facilities of Boost [6] allowed us to provide meaningful error messages and parameter validation.

5.3.1 rec_fixed_itr implementation

Using operator overloading and template metaprogramming we have implemented the `rec_fixed_itr` class as a model of the recursively fixed iterator concept.

Using the operator overloading features of C++ it is straightforward to provide operator*, operator->, operator[], and the non-mutating pointer arithmetic operators for `rec_fixed_itr`. Such implementation techniques are common practice and have been well documented in libraries such as Loki [2] and Boost [6]. The C++ `const` keyword allows variables to be made read-only.

The difficulty in implementing rec_fixed_itr occurs in deciding the return types to give to operator*, operator->, and operator[]. When they are providing access to memory that is in the structure of a `rec_fixed_itr` instance their return type must prevent writes and deletes. When they are providing access to memory structure they must mutable. For this, we must be able to determine at compile time if a memory dereference will access the contents or the structure of a `rec_fixed_itr` instance.

We use C++ template metaprogramming to accomplish this. Originally, the C++ template instantiation mechanism was intended to provide a type-safe alternative to macros for the generation of families of functions that differ only in their types. But then Erwin Unruh [53] discovered that the C++ template instantiation mechanism offered both branching and recursion, allowing arbitrary computation at compile time. The C++ template instantiation mechanism offers a

Turing-complete functional language that executes at compile time.

The presence of a Turing-complete programming language in the C++ compilation process has been widely used in libraries such as Boost [6], Loki [2], Blitz++ [54, 15], and GMCL [15, 14] to provide both optimizations and improved client interfaces. There are several excellent books [1, 2, 14] on library design using template metaprogramming in C++.

Definition 5.3.22. A C++ metafunction is a C++ class template that takes its parameters as template arguments and returns types or integral constants as nested members.

The key to determining if a memory dereference will return an element of the struct lies in determining the dimension of the variable returned by the dereference. Since the dimension of a variable is determined by its type, we can reduce the problem to computing the dimension of an arbitrary type. Figure 5.3.2 shows an implementation of the template metafunction `Dim` we use to compute the dimension of a type. The function is defined recursively. The dimension of the `NullType` is defined as zero and used as the base case of the recursion. The `NullType` is a placeholder type used to represent an unusable type, and serves a similar role as null does for pointers. If the argument to the `pointsTo` metafunction is a C++ pointer, it returns the type pointed to by the argument.; otherwise it returns the `NullType`. The `isNullType` metafunction returns `true` if its argument is the `NullType`; otherwise it returns false. The `if_value` metafunction returns its second argument if its first argument is `true`; otherwise it returns its third argument. All metafunctions called by `Dim` were implemented by us for the sole purpose of computing the return types of the pointer operators of the `rec_fixed_itr` concept. In total, 22 metafunctions were required for the computation of all needed return types.

Figure 5.3.3 offers an outline of the implementation of the `rec_fixed_itr`. There are four major techniques to take note of. (1) The constructor is declared without the keyword `explicit`. This allows the compiler to construct `rec_fixed_itr`'s implicitly from variables of the `CType`. As a result, functions declared to take an argument of type `rec_fixed_itr<T*`
`textgreater` may still be passed variables of type T*. (2) The dimension of the `CType` is computed at template instantiation and embedded in the `rec_fixed_itr` type. This allows the dimension to be passed, via the typedef `self`, to the metafunction `return_type` for computation of the re-

```
//Metafunction: Dim
//Arguments:    T - a type
//Description:  Computes the dimensionality of T.
//Returns:      value - dimension of T
template<typename T> struct Dim;

      template<> struct Dim<NullType>{
         enum { value = 0 };
      };//struct Dim<NullType>

      template<typename T>

      struct Dim{
         enum {

            value = if_value<
               isNullType<
                  typename pointsTo<T>::type
               >::value,
               Dim<NullType>::value,
               1 + Dim<typename pointsTo<T>::type>::value
            >::value

         };//enum

      };//struct
```

Figure 5.3.2: The metafunction Dim computes the dimensionality of a type.

turn type. (3) The `proxy_` member is always wrapped in an object of a type computed by the `return_type` metafunction. This allows the `proxy_` to be used in the `explicit` construction of other types. The metafunction `return_type` returns as types both `rec_fixed_itr` with a lower dimension than the current `CType`, const references, and unprotected references. This implicit construction of return values allows the `return_type` to compute types that will protect the structure of the memory accessible from `proxy_` while at the same time allowing access to the contents.

Note: The actual `rec_fixed_itr` implementation is significantly more complicated. All told, the `rec_fixed_itr` and its 22 supporting metafunctions comprise 700 lines of code. The full implementation handles all pointer arithmetic operations required to support the full range of pointer functionality specified by the recursively fixed iterator concept and validates that `rec_fixed_itr` is only instantiated with valid template arguments.

5.4 simple_ptr

In the process of porting SACLIB from C to C++ we considered converting all return values to be passed by reference. As references do not support pointer arithmetic, this would have clearly indicated from the prototype that no pointer arithmetic would be performed on the function arguments intended to be used only as return values. However, this would have resulted in the need for all client to be altered so as to pass an object or variable to store the return value instead of an address.

To maintain backwards compatibility with existing clients, we implemented a wrapper type, `simple_ptr`, that was a model of the STL's trivial iterator concept [4]. Objects of type `simple_ptr` provide an operator->for member access and operator* for dereferencing. However, they do not provide any of the operators needed for pointer arithmetic. By changing the type of all pointers intended solely for return values to `simple_ptr`s successful compilation of SACLIB guarantees that pointer arithmetic is not performed on the arguments intended to be used solely as return values. This is because the lack of operators for pointer arithmetic causes any attempt to use them to result in a compile time error.

It turned out that SACLIB had correctly avoided the use of pointer arithmetic on all function

```
template<
   typename     CType,
   unsigned int Dimension = Dim<CType>::value
>
class rec_fixed_itr{

   public:

      typedef rec_fixed_itr<CType,Dimension> self;

      //ctor

         rec_fixed_itr(const CType proxy):
            proxy_(proxy)
         {};//rec_fixed_itr

      //example of return type computation

         typedef
            typename return_type<self>::op_star_type
            op_star_ret
         ;

         inline op_star_ret operator*(){
            return op_star_ret(*proxy_);
         };//operator*

   private:

      CType proxy_;

};//rec_fixed_itr
```

Figure 5.3.3: Outline of `rec_fix_itr` implementation technique

arguments intended to be used as return value pointers. The introduction of the `simple_ptr` wrappers did not detect or remove and defects from SACLIB, but it did prevent the introduction of pointer arithmetic defects as future maintenance is performed.

5.5 Proving the Memory System Leak Free and Double Delete Free

For programs using the `array` classes of of Section 5.1 and 5.2 and the `rec_fixed_itr` classes of Section 5.3 for memory management, the absence of resource leaks and double deletes can be validated by the compiler at compile time. We will use the definitions and concepts from the STL that are reproduced in Section 5.3. To prove this claim we define the following relevant static properties.

Definition 5.5.1. Let X be a C++ type. *Implicitly acquired resources of* X are any resources X guarantees to manage correctly for its clients. X is a model of the *Leak Free Destroyable* concept if for all instantiations x of objects of type X the method $X::\tilde{}X$ correctly releases all resources that were implicitly acquired by x and it is statically verifiable that there is no execution sequence of x's methods that can produce a resource leak.

Remark 5.5.2. Inspection of the three functions using `new`/`delete` (˜100 lines of code) clearly shows the `array` classes are models of Leak Free Destroyable. Also, built-in C++ types and C-style structs of built-in C++ types are models of Leak Free Destroyable.

Definition 5.5.3. Let X be a C++ type. X is a model of the *Double Delete Free Destroyable* concept if for all instantiations x of objects of type X the method $X::\tilde{}X$ does not cause multiple deletions of resources implicitly managed by X and it is statically verifiable that there is no execution sequence of x's methods that can produce a double deletion of resource implicitly managed by X.

Remark 5.5.4. Inspection of the three functions using new/delete (˜100 lines of code) clearly shows the `array` classes are models of Double Delete Free Destroyable. Also, built-in C++ types and C-style structs of built-in C++ types are models of Double Delete Free Destroyable.

Definition 5.5.5. Let X be a C++ type. X is a model of the *Safely Manageable* concept if and only if it is a model of the Leak Free Destroyable and Double Delete Free Destroyable concepts.

Definition 5.5.6. Let C be a C++ type that models one of the container concepts from the STL. Then, for all instances c of type C, the elements contained by c are implicitly managed resources. If C is also a model of the Safely Manageable concept, and it contains elements of a type that models the Safely Manageable concept then C is a model of the *Resource Safe Container* concept.

Remark 5.5.7. SACLIB was originally written in C. Therefore, it was not possible for it to have C++ classes. The new SACLIB does not introduce C++ classes that are stored in `array`s. Hence, all types the new SACLIB stores in `array`s are built-in C++ types and C-style structs of built-in C++ types. As a result, all `array`s in the new SACLIB are models of the Resource Safe Container concept.

Theorem 5.5.8. *Let C be a C++ type that is a model of the Resource Safe Container concept. Then all instances c of type C will not leak or double delete resources.*

Proof. Immediate from Definition 5.5.6. □

Theorem 5.5.9. *Let P be a C++ program. Let P use Resource Safe Containers for all resource management. Let all functions in P be const-correct. Let all fully pointer-like function arguments in P be passed by types that are models of the recursively fixed iterator concept. Then, P never leaks or double deletes resources.*

Proof. Resource safe containers guarantee the resources managed by the container will not be leaked or double deleted. By Theorem 5.3.20, all functions in P will not leak or double delete resources referred to by their fully pointer-like arguments because the types of all fully pointer-like arguments are models of the recursively fixed iterator concept. □

Remark 5.5.10. The new SACLIB uses the one dimensional array classes for all one dimensional memory management. By Remark 5.3.10, all functions in the new SACLIB are const respecting. Therefore, the new SACLIB does not contain memory leaks or double deletes arising from managing

one dimensional memory. Of the over 1,000 functions contained in SACLIB, 8 occurrences of the old memory management system have not been replaced due to the need for manual installation of the new memory management. This is not a defect in the memory management system, but rather the 8 functions are the ones that could not be mechanically transformed.

6. Performance Testing

In order to obtain the most accurate timing possible, hardware performance counters were used to measure the number of CPU cycles required for different methods of memory management. All measurements were obtained on a 3.0 GHz Pentium 4 (1024KB L2 cache) running Fedora Core 2. The Performance Counter API (PAPI) [26] version 3.0.7 was used to collect all measurements. The single processor 2.6.5 kernel shipped with Fedora Core 2 was patched with the perfctr-2.6.x patch distributed with PAPI. Tests were compiled with g++ version 3.3.3 [17] and icc [27] version 8.1. Both compilers were given the flags "-O3 -march=Pentium4 -DNDEBUG".

Experiments were repeated 500 times for each compiler. Cycle counts were obtained with `PAPI_get_virt_cyc`. Timings were discarded if they were more than twice the mean. Thirty-one measurement were discarded for gcc; 40 were discarded for icc. Average times are reported in Table 6.1. The icc results clearly show compilers can avoid any abstraction penalty from our class interface; the gcc results show not all compilers do. Both results show our specialized classes perform better than the general purpose `std::vector`.

icc 8.0 benchmarks				
1-dim	alloc	read	write	dealloc
pointer	9357.3	1554.2	2785.6	5734.1
array	9359.7	1561.6	2786.8	5782.2
array_stack	931.2	1564.9	2751.3	223.6
vector	10867.1	1521.2	2776.9	6964.7
2-dim	alloc	read	write	dealloc
pointer2d	25046.8	2377.5	5108.0	13022.8
array2d	21951.6	1998.9	5193.9	13409.8
vector2d	35301.1	3524.0	6314.1	20654.3

gcc 3.3.3 benchmarks				
1-dim	alloc	read	write	dealloc
pointer	7081.1	6317.5	7249.7	4581.6
array	7079.6	7294.0	7252.3	4679.5
array_stack	986.0	7340.0	7251.5	203.1
vector	10279.0	7315.1	7269.3	5863.0
2-dim	alloc	read	write	dealloc
pointer2d	17644.2	7719.1	7741.9	12023.0
array2d	20466.1	7856.7	9248.5	12911.0
vector2d	31682.2	8073.4	11844.6	17039.2

Table 6.1: Machine cycles for various types of memory management. *Pointer* designates the non-garbage collected memory management used by the old SACLIB, *array* stands for our management of heap-allocated memory, *array_stack* for our management of memory that is allocated (more efficiently) on the system stack, and *vector* refers to the vector-class available in STL. The table reports the average number of machine cycles required for allocating and deallocating 500-elements, and for reading and writing 500 elements.

7. Writing Code for the Optimizer

At the machine code level, programmers desire the same functionality from the `rec_fixed_itr` class as they do from built in pointers. This expectation is based on the fact that the at its core, the `rec_fixed_itr` must perform the same pointer operations required of pointer. Internally, all of the pointer operators of the `rec_fixed_itr` class are implemented using pointers.

Because of this, any performance overhead suing the `rec_fixed_itr` class must be a result of the compiler's inability to determine that the machine code required for a `rec_fixed_itr` is identical to that required for a pointer. The `rec_fixed_itr` was carefully designed to give compiler every opportunity to determine this equivalence.

There are three critical aspects to the `rec_fixed_itr` that allow for optimizing compilers to produce efficient machine code for the `rec_fixed_itr`:

1. Template metaprogramming is only used for return type computations.
2. All operators are implemented as inline forwarding functions.
3. Where ever possible, operator arguments are taken as `const` qualified references.

By restricting template metaprogramming to return type computations, there will be no code generated by the template metaprogramms. Because the type computation must be completed before the compiler knows the type, code generation cannot begin until the type computation has completed. This precludes the situation of code being passed to the optimizer from a code generating template metaprogam. While optimizers frequently can remove the overhead from template generated code, some generators [10] produce code that optimizers cannot remove all unessential overhead from. Type only metafunctions side step this problem.

Implementing the operators as `inline` and using `const` qualified references works together to help the optimizer. By making all functions `inline`, the compiler is allowed to see more of the calling context of the operators. By making the references `const` qualified, the compiler is assured writes will not occur to the argument. This saves the compiler from the task of having to try and

determine this information from context. Giving the compiler access to this information is essential for it to be able to prevent extraneous overhead that could otherwise be introduced through the used of operator overloading.

This design makes performance hinge upon optimizations that modern compilers are reasonably competent at performing. However, as the performance measurements show, not all compilers are capable of recognizing or exploiting the information exposed by this design.

8. Automatic Conversion

The old SACLIB was written in K&R C [31]. In K&R C functions can be defined as

```
void function(a,b)
   int a;
   double b;
{

}
```

While this is a legal function definition in both K&R C and ANSI C [28, 7], it is not a legal function definition in ANSI C++ [29, 49]. In ANSI C++, the function must be defined as

```
void function(int a, double b){

}
```

Additionally, K&R C requires only the return type of a function to be known prior to allowing it to be called. The above function would have its return type described as

```
void function();
```

ANSI C allows a full function prototype that specifies both the return type of the function and the types of the arguments

```
void function(int a, double b);
```

ANSI C++ requires the use of a full function prototype.

SACLIB provided full function prototypes for compilers that were ANSI C compliant. The function prototypes were wrapped in a macro

```
void function P__((int a, double b));
```

where P_ was defined as

```
#ifdef __STDC__

   #define P__(A) A

#else

   #define P__(A) ()

#endif
```

This ensured that function prototypes were only used by ANSI C compilers. We first compiled SACLIB using an ANSI C compiler to ensure the correctness of the prototypes. Once we knew the prototypes matched, a perl script was written that read the existing prototypes, took the arguments from the prototypes and placed them in the function definitions, and deleted the old K&R function argument definitions. Another perl script was then used to the P_ macros.

As SACLIB was a generally well-structured C program, this was all that was required to port SACLIB from C to C++.

9. Conclusion

We have shown how several features of the C++ language can be combined with our theory to ensure correct allocation and deallocation of memory. User extensibility of the C++ type system, operator overloading, and template metaprogramming allow the compiler to verify the static properties that form the preconditions of correct memory management.

As a result, SACLIB 3.0 and any future extensions using our memory subsystem will be leak and double delete free. We have significantly improved the self documenting nature of the SACLIB code base and substantially reduced the likelihood of memory errors. Consequently, programmers will spend less time detecting, diagnosing, and repairing memory defects.

We show that static typing can be combined with concepts and metaprogramming to provide correct memory management. We have shown that safety properties can be specified in concepts. We expect this strategy can be used for other safety properties to provide similar correctness, maintenance, and readability benefits as we have obtained for memory management.

Our experiments show that these benefits can be obtained without degradation of performance.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison Wesley Longman, Inc., 2005.

[2] Andrei Alexandrescu. *Modern C++ Design*. Addison Wesley Longman, Inc., 2001.

[3] Americain National Standards Institute, `http://ansi.org/`. *Information Technology - Programming Language - Common Lisp (formerly ANSI X3.226-1994 (R1999))*, 1999.

[4] Matthew H. Austern. *Generic Programming and the STL*. Addison Wesley Longman, Inc., 1998.

[5] Hans-J. Boehm. The space cost of lazy reference counting. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 210–219. ACM Press, 2004.

[6] Boost. Boost libraries and documentation. `http://www.boost.org`.

[7] Standards Institute British. *The C Standard: Incorporating Technical Corrigendum 1*. John Wiley and Sons, September 2002.

[8] Christopher W. Brown. QEPCAD B: A program for computing with semi-algebraic sets using CADs. *SIGSAM Bulletin*, 37(4):97–108, 2003.

[9] Christopher W. Brown. QEPCAD B: A system for computing with semi-algebraic sets via cylindrical algebraic decomposition. *SIGSAM Bulletin*, 38(1):23–24, 2004.

[10] Dov Bulka and David Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley Professional, 1999.

[11] George E. Collins et al. `SACLIB` User's Guide. Technical Report 93-19, Research Institute for Symbolic Computation, RISC-Linz, Johannes Kepler University, A-4040 Linz, Austria, 1993.

[12] George E. Collins and Hoon Hong. Partial CAD contruction in quantifier elimination. Technical Report OSU-CISRC-10/89 TR45, Computer Science Department, Ohio State University, 1989.

[13] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, New York, NY, USA, 2003. ACM Press.

[14] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.

[15] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[16] Electric Fence Developers. Electric Fence. `http://www.pf-lug.de/projekte/haya/efence.php`.

[17] GNU. The GNU Compiler Collection. `http://gcc.gnu.org/`.

[18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The*. Addison-Wesley Professional, 3rd edition, 2005.

[19] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[20] Douglas Gregor, Sibylle Schupp, and David Musser. Design patterns for library optimizations. *Scientific Programming*, 11:309–320, 2003.

[21] Matthew Harren and George C. Necula. Lightweight wrappers for interfacing with binary code in ccured. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003. Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 209–225. Springer-Verlag, 2004.

[22] H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 261–264. ACM Press, 1990.

[23] Hoon Hong, Richard Liska, and Stanly Steinberg. Testing stability by quantifier elimination. *Journal of Symbolic Computation*, 24(2):161–187, 1997.

[24] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The design of the SACLIB/PACLIB kernels. *Journal of Symbolic Computation*, 19(1–3):111–132, 1995.

[25] IBM Corporation. Rational Purify. `http://www-306.ibm.com/software/awdtools/purify/`.

[26] Innovative Computing Laboratory. PAPI. `http://icl.cs.utk.edu/PAPI`.

[27] Intel Corporation. The Intel C++ Compiler. `http://www.intel.com/software/products/compilers/clin/`.

[28] International Standards Organization, `http://www.iso.org`. *ISO/IEC 9899:1999: Programming languages—C*, 1999.

[29] International Standards Organization, `http://www.iso.org`. *ISO/IEC 14882:2003: Programming languages—C++*, 2003.

[30] Nicolai M. Josuttis. *The C++ Standard Library*. Addison Wesley Longman, Inc., 1999.

[31] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1st edition, 1978.

[32] Werner Krandick and Kurt Mehlhorn. New bounds for the Descartes method. *Journal of Symbolic Computation*, to appear.

[33] Yukihiro Matsumoto. *Ruby In A Nutshell*. O'Reilly, 2001.

[34] Scott Meyers. *Effective C++*. Addison Wesley Longman, Inc., 2nd edition, 1997.

[35] Scott Meyers. *Effective STL*. Addison Wesley Longman, Inc., 2001.

[36] David Musser, Sibylle Schupp, and Rüdiger Loos. Requirement oriented programming concepts, implications, and algorithms. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 12–24. Springer-Verlag, 2000.

[37] David R. Musser and Zhiqing Shao. Concept use or concept refinement: An important distinction in building generic specifications. In C. George and H. Miao, editors, *Formal Methods and Software Engineering: 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002. Proceedings*, volume 2495 of *Lecture Notes in Computer Science*, pages 132 – 143. Springer-Verlag, 2002.

[38] GC Necula, J Condit, M Harren, S McPeak, and W Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[39] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[40] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, 1972.

[41] Open Directory Project. The open directory project. `http://dmoz.org/Computers/Programming/Languages/Garbage_Collected/`.

[42] Guido Van Rossum and Jr. Fred L. Drake. *Python Language Reference Manual, The*. Network Theory, 2003.

[43] W. Schreiner, W. Danielczyk-Landerl, M. Marin, and W. Stöcher. A generic programming environment for high-performance mathematical libraries. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 256–267. Springer-Verlag, 2000.

[44] S. Schupp, D. Gregor, D. Musser, and S.M. Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, 2002.

[45] S. Schupp and R. Loos. SuchThat—Generic programming works. In M. Jazayeri, R. G. K. Loos, and D. R. Musser, editors, *Generic Programming: International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 133–145. Springer-Verlag, 2000.

[46] Thomson Scientific. Web of science. `http://www.isinet.com/`.

[47] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison Wesley, 1994.

[48] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison Wesley, 2000.

[49] Bjarne Stroustrup. *The C++ Standard: Incorporating Technical Corrigendum No. 1*. John Wiley and Sons, 2003.

[50] Herb Sutter. *Exceptional C++*. Addison Wesley Longman, Inc., 1999.

[51] Herb Sutter. *More Exceptional C++*. Addison Wesley Longman, Inc., 2001.

[52] Valgrind Developers. Valgrind. `http://valgrind.kde.org`.

[53] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison Wesley Longman, Inc., 2003.

[54] Todd L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R.R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, 1998.

[55] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.