

# Static Detection of Leaks in Polymorphic Containers

David L. Heine  
Tensilica, Inc.  
dlheine@tensilica.com

Monica S. Lam  
Computer Systems Laboratory  
Stanford University  
lam@stanford.edu

## ABSTRACT

This paper presents the first practical static analysis tool that can find memory leaks and double deletions of objects held in polymorphic containers. This is especially important since most dynamically allocated objects are stored in containers.

The tool is based on the concept of object ownership: every object has one and only one *owning* pointer. The owning pointer holds the exclusive right and obligation to either delete the object or to transfer the obligation. This paper presents a new type system that allows different instances of a polymorphic container to hold different types of elements, and to independently own or not own their elements.

Our tool is sound: it will report all potential memory leaks and multiple deletions of pointers in a program. Our system automatically identifies the container implementation routines in an application. The user provides a short specification on the container structure and ownership constraints for these routines. The system then solves for the ownership constraints flow- and context-sensitively, and reports inconsistencies in ownership constraints as potential memory leaks and double deletions.

We applied our tool to a suite of five large open-source and commercial C and C++ applications totaling one million lines of code. The tool successfully identified memory leaks in these programs and found double deletions of objects that could lead to program failures or security vulnerabilities.

## Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.4 [Software Engineering]: Program Verification; D.3.4 [Programming Languages]: Processors—*Memory Management*

---

This material is based upon work supported in part by the National Science Foundation under Grants No. 0086160 and 0326227.

## General Terms

Algorithms, Experimentation, Languages, Verification.

## Keywords

Program analysis, type systems, memory management, error detection, memory leaks.

## 1. INTRODUCTION

Container data structures, such as lists, stacks, maps, and arrays, are commonly used to hold references to many dynamically allocated objects. Containers are usually designed as *generic* or *polymorphic* data structures capable of holding objects of any data type. Because a majority of dynamically allocated objects are held in containers, it is important for memory leak detection tools to find leaks in the use of polymorphic containers. This paper presents the first tool based on static analysis that can find leaks in containers with polymorphic element ownership in large C and C++ applications.

### 1.1 Monomorphic Object Ownership

Our memory leak detection technique is based on the concept of object ownership, a notion often used by programmers in managing memory. We previously provided a formalization of object ownership and used it as the basis for an automatic static memory leak detector called Clouseau [21]. Objects can be pointed to by multiple pointers, but one and only one of these is considered to be the *owning* pointer. The owning pointer holds the exclusive right and obligation to either delete the object or to transfer the right to another owning pointer. Ownership can be passed through local variables, passed into procedures, and returned from procedures until an object is freed. In this monomorphic model, a monomorphic *ownership type* is associated with each pointer and the field of each C++ class. This is inadequate for handling polymorphic containers.

### 1.2 Polymorphic Object Ownership

In practice many containers are designed to have *polymorphic object ownership*: that is, some instance of a container type may own its elements, while some other does not. The previous analysis generated false warnings in the presence of two distinct instances of the same container type that separately hold owning and non-owning references to their elements. Containers are complicated by the fact that they can be *nested*; the elements in a container may themselves be containers. As a pointer to a container is passed around,

ownership of the container may transfer from one pointer to another. However, the structure of the container's elements and the ownership of its elements remains invariant. This property holds for each level of nesting for containers of containers. Different instances of containers, i.e. containers allocated at different program points, can have elements with different structure or ownerships.

### 1.3 Automatic Inference with User Specification

It is difficult to determine polymorphic container types in C programs. They are often written to contain elements with a generic `void *` type that can be used to hold pointers, function pointers, or even some scalar integer types. C++ programs, written using templates, are easier to handle. In either case, pointer manipulation can obscure programmer intent and make a program difficult to analyze.

Our system can automatically detect the *container implementation routines* that manipulate the internals of a container. We require users to specify *interfaces* for these routines that summarize the polymorphic container and ownership behavior of their *parameters*, both inputs and outputs. This is practical because programmers only need to document a small number of container implementation routines. Well-engineered applications use only a small library of container routines, the C++ STL library being one such popular example. Specifications can be provided once for a library and can be reused across all applications using the library.

Given the specification on container implementation routines, our system automatically infers the container structures and ownerships of the rest of the program in a flow- and context-sensitive manner. If the system finds the program to be type-safe with respect to the container structure and ownership properties, then there are no memory leaks nor double deletes. Otherwise, the inconsistencies found correspond to all the potential memory leaks and double deletes in the program.

### 1.4 Evaluation

Static memory leak detection tools have previously been evaluated by the number of errors found and the ratio of false warnings to true errors reported. This does not report how many leaks go undetected. A leak analysis should be evaluated by the volume of memory leaks detected; finding leaks that cause large dynamic leak volumes is better than finding leaks that occur only on rare error conditions. We use Valgrind [24] to dynamically measure the volume of memory leaked by a program. Evaluation with dynamic tools is useful, but provides an incomplete picture of memory leaks since static analysis can find errors that may not occur in a particular run. We evaluate our analysis tool based on four criteria: the number of program errors identified, the percentage of warnings that help identify errors, the volume of leaks removed from a program by fixing the identified errors, and the number of program errors identified that would not have been found by only using dynamic techniques.

We have implemented all the ideas presented in this paper as an extension to the original Clouseau system. To get a realistic evaluation, we applied our tool to five large C/C++ applications, totaling over a million lines of code.

We found containers in all five applications and containers with polymorphic ownership in four. Our analysis was

effective at finding both memory errors and important leaks. We identified 592 errors; including 10 double deletions that could corrupt memory. It performed very well in three of the four applications with dynamic leaks. It was also effective at finding leaks that did not occur in a particular program execution. It found leaks not identified dynamically in all of the applications and 69 memory problems in the application that had no dynamically identifiable memory leaks. For the program with the most leaks, adding polymorphic container support to the analysis more than doubled the effectiveness of our leak detection.

### 1.5 Summary of Contributions

The contributions of this paper include:

1. An object ownership model with polymorphic abstract containers.
2. Use of the monomorphic ownership model to identifying container implementations automatically.
3. A language for describing polymorphic abstract container structure and ownership.
4. Use of a standard polymorphic type inference to find nested container structures based on user specifications.
5. A context-sensitive and flow-sensitive polymorphic container ownership inference algorithm.
6. An implementation of a tool to detect leaks in polymorphic containers.
7. Experimental validation of the effectiveness of our techniques to identify polymorphic containers and identify large volumes of memory leaks.

### 1.6 Paper Organization

We first review the basic ownership model in Section 2. We then present a polymorphic stack in Section 3 that we use as a running example throughout the paper. Section 4 defines the system of types and constraints we use to summarize the polymorphic ownership of each function. Section 5 presents our more user-friendly language that allows users to express the interfaces of container implementation routines. Section 6 then describes how our system automatically infers the interfaces of all other routines and presents warnings to the user. Section 7 reports on the experimental results. We discuss related work in Section 8 and conclude in Section 9.

## 2. THE MONOMORPHIC OWNERSHIP MODEL

We start by reviewing the basic monomorphic ownership model that does not include support for polymorphic containers; details have been presented elsewhere [21]. The model is based on three properties:

**Property B1.** *Only an allocator can return a pointer with new ownership.*

**Property B2.** *There exists one and only one owning pointer to every object allocated but not deleted.*

**Property B3.** *A deallocator, applicable only to an owning pointer, renders the pointer non-owning.*

These properties guarantee that no object is deleted more than once. They also guarantee that a program has no memory leaks except in the case where objects may participate in a cycle of owning references [21]; however, we do not know of any usage of the ownership model that would create such cycles.

## 2.1 Constraint-Based Type System

We have previously introduced an ownership constraint-based type system such that type-safe programs satisfy all the above properties, and therefore have no leaks or double deletes [21]. The type system has an inference rule for each kind of statement specifying the constraints that must be satisfied to guarantee type-safety. We describe the inference rules informally below.

For an allocation statement `p = new <type>`, we require that `p` must not be an owning pointer before the statement but an owning pointer afterwards.

For a deallocation statement, `delete p`, `p` must be an owning pointer before the statement and not owning afterwards.

For a pointer assignment statement `q = p`, ownership must be conserved, namely:

1. `q` must not be an owning pointer before the statement,
2. if `p` is an owning pointer before the statement, the ownership may be kept with `p` or transferred to `q`,
3. if `p` is not an owning pointer before the statement then neither `p` nor `q` is owning.

In parameter passing, ownerships may also be optionally transferred from the actual to the formal arguments, they are modeled in a manner similar to assignment statements.

In this basic model, we assumed that indirectly written pointers cannot hold ownership in C. That is, given an indirect store statement `*p = q`, we assume that `*p` is not an owning pointer. In C++, a member must either be owning or non-owning at all boundaries of public method invocations. The basic model cannot handle polymorphic containers.

Our analysis is flow-sensitive but path-insensitive. The ownership of a pointer can vary from program point to program point, but every pointer in all instances of the same program point must have the same ownership regardless the path taken to get there.

Our analysis is fully context-sensitive. Parameters at different call sites can have different ownerships provided that the constraints of the called routines are satisfied.

## 2.2 Ownership Constraints

We have developed an inference algorithm that collects ownership constraints from a program and infers an interface that summarizes the constraints for each function. The interfaces associate an ownership type  $\rho$  with each pointer parameter in a function. Ownership types can be assigned either value 1 to indicate it is an “owning” pointer, or 0 to indicate “non-owning”. Constraints used in the interface can be of two different forms, as explained below.

**Linear constraints**, a limited form of 0-1 integer linear programs, are used to model optional ownership transfer. For example, let  $\rho_p$  and  $\rho_q$  be the ownerships of pointer variables `p` and `q` before an assignment statement, and  $\rho'_p$  and  $\rho'_q$  be the ownerships afterwards, the ownership constraint of the assignment can be expressed as:

$$\rho_q = 0 \wedge \rho_p = \rho'_p + \rho'_q$$

**Instantiation constraints** are used to handle context-sensitive ownerships. We require that the ownership type of an actual parameter  $\rho'$  at each call site  $i$  be an *instantiation* of a formal parameter  $\rho$ , written  $\rho' \preceq^i \rho$ . When actual

ownerships are substituted for the formal ownerships in a constraint, the resulting constraint must still be satisfiable.

Consider the identity function that returns its input unchanged. While one invocation may pass in an owning pointer and get back an owning pointer, another invocation may pass in a non-owning pointer and get back a non-owning pointer. A context-insensitive approach would require actual parameters at each call site to have the same ownership and would generate very imprecise results.

The identity function has the constraint  $\rho_p = \rho_q$  where  $\rho_p$  and  $\rho_q$  are the ownerships of the input and output, respectively. Let the ownerships of the actual argument and return at a call site be  $\rho'_p$  and  $\rho'_q$ . Constraints on the actuals can be more specific than constraints on the formals. Thus  $\rho'_p = 1 \wedge \rho'_q = 1$  or  $\rho'_p = 0 \wedge \rho'_q = 0$  would be valid constraints on the actuals.

## 2.3 Ownership Type Inference

To determine if there are errors in the program, our ownership inference algorithm collects all the constraints from the program and tries to find an assignment to the ownership variables such that all the constraints can be satisfied. If the constraints are consistent, then there are no double deletes or memory leaks; otherwise, our algorithm generates warnings on potential errors.

For the simple example below:

```
p = new int;    (1)
q = p;         (2)
delete q;      (3)
```

our type system generates the constraint that `p` must be an owning pointer after statement (1), `q` must be owning before statement (3). All the constraints can be satisfied by assuming that the assignment in statement (2) transfers ownership from `p` to `q`. Therefore, there are no leaks or double deletes. Note that ownership is a property of pointer variables. This approach eliminates the need for counting references [21].

The ownership inference algorithm is fully context-sensitive. Using a fixpoint calculation, it computes an interface for each function that summarizes the ownership constraints between its parameter arguments and return values. The interface represents the satisfying assignments of 0,1 values to the ownerships of arguments and return values. The algorithm is sound in that it finds all the constraint violations and hence all potential leaks.

It is important that we correctly identify the source of errors. Our technique gives higher priorities to more precise constraints and considers them first. For example, constraints associated with deallocations are given a higher priority over constraints requiring that indirectly written pointers must be non-owning. If there is an inconsistency between these two rules, it is likely to be caused by the latter. Our system considers the constraints in decreasing order of priority; a constraint found to be inconsistent with the constraints collected so far is deemed the source of the error, flagged as an error, and excluded from further consideration. The priority of the errors are also reported so users can concentrate on fixing high-priority errors. In addition to the prioritization of constraints, the analysis reports related constraint violations together. We found the algorithm to be effective in pinpointing the source of problems.

## 2.4 Identifying Container Implementation Routines

The monomorphic ownership type system will flag all cases where ownership disappears into memory locations that are written indirectly, array elements, fields in C structures, and non-owning fields in C. Many of these reported errors are false warnings; however these errors are useful because they identify those functions that manipulate owning pointers. These warnings thus serve as indications of where the container implementation routines are and which routines can use manually supplied specifications.

## 3. EXAMPLE: A POLYMORPHIC STACK IN C

Shown in Figure 1 is a standard C declaration and brief description of a stack and a set of routines implementing it. As written, any instance of the stack can hold elements of any type. We assume in this paper that each instance of a stack only holds elements of a single type.

```
typedef struct stack_t {
    int size;        // element count
    void **arr;     // array of elements
} stack;

stack *stack_alloc();
    Allocates a new stack.
void stack_push(stack *s, void *e);
    Pushes an element onto the stack.
void *stack_pop(stack *s);
    Removes an element from the stack.
void stack_free(stack *s);
    Frees the stack without freeing the elements.
void stack_free_all(stack *s, void (*f)(void *));
    Invokes a function f on each element of the stack to free
    the storage associated with the element then frees the
    stack. This is useful for implementing a polymorphic
    stack data structure whose elements may themselves
    be containers.
```

Figure 1: C Interface of a Polymorphic Stack

Our ownership model assumes that any instance of a stack can choose to own or not own its elements, but the choice remains the same throughout the lifetime of that instance. An example illustrating the use of polymorphic ownership is shown in Figure 2. The example has two stacks: `s` points to a stack of integers, and `ss` points to a stack of stacks of integers. This example has no memory leaks because line (7) frees the stack pointed to by `ss` and all its stack members, and line (8) frees the integer pointed by `p`. Reasoning in terms of ownership, we say that the stack pointed to by `s` does not own its members, whereas the stack pointed to by `ss` does. When `p` is pushed onto `s`, `p` retains ownership, and is responsible for its deletion, which happens in line (8).

## 4. POLYMORPHIC OWNERSHIP INTERFACES

Ownership inference summarizes every function in the program with an interface that specifies *constraints* on the

```
void nested_stack_of_ints() {
    /* build a stack of ints */
    stack *s = stack_alloc();      (1)
    int *p = new int;              (2)
    stack_push(s, p);              (3)

    /* build a stack of stacks of ints */
    stack *ss = stack_alloc();     (4)
    stack_push(ss, s);             (5)

    /* free ss, s, p */
    void (*f)(stack *) = &stack_free; (6)
    stack_free_all(ss, f);         (7)
    delete p;                      (8)
}                                   (9)
```

Figure 2: Polymorphic stack usage

*types* of the parameters. Types in the polymorphic ownership model are more elaborate. We need to track the ownership of pointers and, for pointers pointing to containers, the structure of the containers and the ownership designation for each field at each nesting level. We first define the types, then the constraints allowed on the types.

### 4.1 Types

A type  $\tau$  can be one of:

- A type variable  $\alpha$ .
- A pointer type  $(\text{ptr}_\rho \tau)$  which has an ownership type  $\rho$  and a pointed-to type  $\tau$ .
- A parameterized container type  $\beta(\tau)$ . Here,  $\beta$  is the name of the parameterized container, and  $\tau$  is the type parameter. Each container has a number of typed fields  $f: \tau$ .
- A product type  $(\tau \times \tau)$  for representing anonymous aggregates.
- A **base** type representing non-owning integer and floating point scalars.
- A non-owning imprecise bottom type  $\perp$  to represent any object that contains no owning pointers. We actually maintain a disjoint sum of types to better handle function pointers as in [13].
- A function type  $(\tau \rightarrow \tau)$ .
- An empty type **unit** used for empty function parameters.

Fields in containers are abstract. We use a field to represent a set of objects that have the same structure and ownerships. For example, we can declare the **stack** container as:

$$\text{stack}(\tau) \{ \text{elem} : \text{ptr}_1 \tau \}$$

We use the abstract `elem` field to represent the `size` and the `arr` array in the **stack** implementation. The user of the container can treat the stack as having a single pointer that points to an object with parametric type  $\tau$ . The pointer `elem` itself is owned by the container, meaning that it will be deleted when the container is deleted.

Our type system is flow-sensitive because a variable may have different types at different program points. Figure 3 shows the types of the variables for our example in Figure 2.

Variable	Type	Lines
<b>p</b>	$\text{ptr}_1 \text{base}$	2 – 8
<b>s</b>	$\text{ptr}_1 \text{stack}(\text{ptr}_0 \text{base})$	1 – 5
<b>s</b>	$\text{ptr}_0 \text{stack}(\text{ptr}_0 \text{base})$	5 – 9
<b>ss</b>	$\text{ptr}_1 \text{stack}(\text{ptr}_1 \text{stack}(\text{ptr}_0 \text{base}))$	4 – 7
<b>f</b>	$\text{ptr}_1 \text{stack}(\text{ptr}_0 \text{base}) \rightarrow \text{unit}$	6 – 9

Figure 3: Variable Types

- **p** is an owning pointer to a base type.
- **s** is a pointer to a stack of non-owning pointers to base-type objects. **s** owns the stack until line 5, where its ownership is transferred into the elements of **ss**.
- **ss** is an owning pointer to a stack of owned stacks of non-owning base-type objects.
- **f** points to a function that accepts an owning pointer to a stack which does not own its elements and returns nothing.

## 4.2 Constraints

The interface of a function places constraints on the types of its parameters. For example, the interface of the function `stack_free` is

$(\text{ptr}_1 \text{stack}(\tau) \rightarrow \text{unit})$  such that  $\text{non-own}(\tau)$ .

This interface says that `stack_free` has one argument and it returns no result. The argument is an owning pointer since it will be freed within the function. It points to a container named `stack` which is parameterized by type variable  $\tau$ . Because `stack_free` does not free any of the elements in the container, the parameterized type must not carry any ownership. This constraint is written  $\text{non-own}(\tau)$  and described below. Constraints on types can be of the following forms:

- Ownership constraints on ownership type components.
- Non-owning constraints ( $\text{non-own}(\tau)$ ). This forces the type  $\tau$  to be either a non-owning pointer variable, a product type with non-owning components, or one of the other (implicitly non-owning) types. These are generated when a variable is overwritten or exits scope.
- Equivalence constraints ( $\tau_1 = \tau_2$ ). Equivalence constraints force two types and their respective components to be equivalent.
- Linear constraints, e.g. ( $\tau_1 = \tau_2 + \tau_3$ ), for modeling assignment statements and parameter passing. As an example, for the statement  $\mathbf{q} = \mathbf{p}$ , we require that  $\tau_p = \tau_p' + \tau_q'$  where  $\tau_p$  is the type of **p** before the assignment, and  $\tau_p'$  and  $\tau_q'$  are the types of **p** and **q** after the assignment, respectively.

If  $\tau_1, \tau_2$ , and  $\tau_3$  are pointer types with ownerships  $\rho_1, \rho_2, \rho_3$ , and pointee types  $\tau_1', \tau_2',$  and  $\tau_3'$ , respectively, then  $\rho_1 = \rho_2 + \rho_3$ , and  $\tau_1' = \tau_2' = \tau_3'$ . If  $\tau_1, \tau_2, \tau_3$  are product types, the linear constraint is defined similarly for each of its components. Otherwise,  $\tau_1 = \tau_2 + \tau_3$  iff  $\tau_1 = \tau_2 = \tau_3$ .

- Instantiation constraints on types ( $\tau_1 \preceq^i \tau_2$ ). These constraints are analogous to the instantiation constraints on ownership types discussed in Section 2.2.

Routine	Type	where
<code>stack_alloc</code>	$(\text{unit} \rightarrow \text{ptr}_1 \text{stack}(\tau))$	
<code>stack_push</code>	$(\text{ptr}_0 \text{stack}(\tau) \times \tau \rightarrow \text{unit})$	
<code>stack_pop</code>	$(\text{ptr}_0 \text{stack}(\tau) \rightarrow \tau)$	
<code>stack_free</code>	$(\text{ptr}_1 \text{stack}(\tau) \rightarrow \text{unit})$	$\text{non-own}(\tau)$
<code>stack_free_all</code>	$(\text{ptr}_1 \text{stack}(\tau) \times (\tau \rightarrow \text{unit}) \rightarrow \text{unit})$	

Figure 4: Stack Implementation Routine Specifications

The interfaces for each of the stack implementation routines are shown in Figure 4. Equivalent types have the same name in a type expression. Type constraints that cannot be expressed implicitly as equivalence are presented in the column labeled “where”.

`stack_alloc` takes no arguments and returns an owning pointer pointing to a parameterized stack container with elements of unconstrained type  $\tau$ .

`stack_push` accepts a non-owning pointer to a stack, whose elements have types equivalent to that of the second argument. By the definition of type equivalence, if the second argument is a pointer, the stack owns its elements iff the second argument is a pointer with ownership. Moreover, if the second argument points to a container, then the elements of the stack and the second argument have the same structure and ownerships at each nesting level.

`stack_pop` accepts a non-owning pointer to a stack of elements with types equivalent to that of the returned value.

`stack_free_all` accepts an owning pointer to a stack, and a function that accepts an argument that has the same type as the element on the stack.

## 5. POLYMORPHIC CONTAINER SPECIFICATIONS

We assume that all the indirect loads and stores involving owning pointers are encapsulated in *container implementation routines*. Conversely, any indirect loads and stores in the rest of the code are assumed to involve only non-owning pointers. The user is responsible for specifying correct interfaces to container implementation routines; our inference algorithm automatically finds the interfaces to all the other routines and reports all potential errors.

Instead of asking programmers to learn the formal type notations defined above, we have created a more user-friendly specification language that is modeled after definitions of templates and generic classes. The language is simply syntactic sugar for a subset of the formal type system. The full language is defined in [20]. To give the reader a flavor of the language, the user-supplied equivalent of the formal specifications for the stack routines are shown in Figure 5.

We use the keywords **container** to declare containers, **own** to stand for an owning pointer, **ref** to stand for a non-owning pointer, **where** for including additional type constraints, and the predicate **ref\_type**(T) to indicate that T is a non-owning type, equivalent to a formal constraint like  $\text{non-own}(\tau)$ . The correspondence of this specification with the formal version is self-explanatory.

```

type T;

container stack<T> {
  own T * elem;
};

own stack<T> *stack_alloc();
void stack_push(ref stack<T> *s, T e);
T stack_pop(ref stack<T> *s);
void stack_free(own stack<T> *s)
  where { ref_type(T) };
void stack_free_all(own stack<T> *s,
  void (*f)(T));

```

Figure 5: Specification for the Stack Container and Interfaces in Figure 1

## 6. POLYMORPHIC TYPE SYSTEM AND INFERENCE

The polymorphic type system is based on the same properties (B1–B3) as the monomorphic version; however, it removes many false warnings and increases the accuracy of others since it properly tracks ownership across polymorphic containers. In addition, to reduce the number of warnings presented to the user, related constraint violations are grouped into a single warning. In this system, we assume the user-provided interfaces to container implementation routines are correct. The type system that we have defined guarantees a type-safe program will not have memory leaks nor double deletes. It is sound: it gives warnings on all potential errors.

### 6.1 Type System

The polymorphic type system definition is similar to the monomorphic model. It is flow- and context-sensitive but path-insensitive. A formal description of our type system is presented in [20]. Here we focus only on the major differences from the monomorphic model.

As before, the type of a pointer can vary from program point to program point, and every pointer in all instances of the same program point must have the same type. The polymorphic system replaces a pointer’s ownership type with a structural type that contains both an ownership type and a structural pointed-to type.

The pointer assignment statement  $q = p$  is handled as follows. Let  $\tau_p$  and  $\tau_q$  be the types of variables  $p$  and  $q$  before an assignment statement, and  $\tau'_p$  and  $\tau'_q$  be the types afterwards. The constraint to be generated is:

$$\text{non-own}(\tau_q) \wedge \tau_p = \tau'_p + \tau'_q$$

$q$  must not carry any ownership before the assignment. The linear constraint requires that the ownerships of pointers are conserved and, after the statement,  $p$  and  $q$  both point to the same type as the pointee of  $p$  before the statement.

All the interesting indirect memory accesses are assumed to be encapsulated in container implementation routines, for which specifications have been supplied. We generate instantiation constraints to ensure that such interfaces are obeyed in a context-sensitive manner. Indirect memory accesses outside of the container implementation routines are assumed to operate on non-owning pointers, like before. All

these inference rules ensure that there is one and only one owning pointer to every object at all times, provided that the user-specified interfaces are correct.

### 6.2 Type Inference

The type inference algorithm consists of two steps, the first infers the structure of the containers, the second solves the constraints on ownerships.

**Structure Inference.** We use a standard polymorphic inference algorithm to determine the structure of containers implied by the the specified interfaces across the program. The problem of solving structure constraints is equivalent to semi-unification [22], which has been proven undecidable [23]. Henglein proposed a semi-decision procedure that has never failed to terminate. Our implementation is based on a more recent algorithm from Fähndrich et al. [13]. Applying the algorithm generates the structural components, but not the ownership components of our types.

**Ownership Inference.** From the structural type inference, we know at each program point whether a pointer points to a container. When it points to a container, we also know the nested structure of that container. We associate an ownership type variable with each component of each nesting level of containers.

The inference required for polymorphic ownership requires many more ownership variables and constraints than a monomorphic ownership system [21]. However, the ownership constraints collected have the same form as those collected in a monomorphic type system. Thus, the same ownership inference algorithm can be used to resolve the ownership constraints. The algorithm pinpoints sources of inconsistency as potential statements that leak memory.

We now return to our polymorphic stack example and show the constraints generated by our type system on this code. Let  $\rho_{p2}$ ,  $\rho_{p3}$ ,  $\rho_{p8}$ ,  $\rho_{s1}$ ,  $\rho_{s3}$ ,  $\rho_{s5}$ ,  $\rho_{ss4}$ ,  $\rho_{ss5}$ , and  $\rho_{ss7}$  be ownership variables for  $p$ ,  $s$  and  $ss$ . Each is subscripted with the statement number where it is generated because of a potential ownership transfer in the program.

From the polymorphic structure type inference, we know that  $s$  is a stack of integers, and  $ss$  is a stack of stacks of integers. Let  $\rho_{s.e}$  and  $\rho_{ss.e}$  be the ownership of elements of the stacks pointed to by  $s$  and  $ss$ , respectively, and  $\rho_{ss.e.e}$  be the ownership of the integer elements in the stacks that are elements of the stack pointed to by  $ss$ .

Let  $\rho_{f.1}$  be the ownership of the function pointer  $f$ ’s first argument.  $\rho_{f.1.e}$  represents the elements of the stack pointed to by the first argument.

The ownership constraints generated by this example are:

Lines 1,2,4: An allocator returns an owning pointer:  $\rho_{s1} = 1$ ,  $\rho_{p2} = 1$ ,  $\rho_{ss4} = 1$ .

Lines 3,5: The specification on `stack_push` says

1. The ownership of the first argument is not passed into the routine:  $\rho_{s1} = \rho_{s3} + 0$ ,  $\rho_{ss4} = \rho_{ss5} + 0$ .
2. The abstract `elem` field has the same ownership as the second argument. This means that (1) if the ownership is passed in, then the `elem` field owns its members  $\rho_{p2} = \rho_{p3} + \rho_{s.e}$ ,  $\rho_{s3} = \rho_{s5} + \rho_{ss.e}$  and (2) if the second argument is a container, then the members of the `elem` field must have the same nested ownerships. From line 5,  $\rho_{s.e} = \rho_{ss.e.e}$ .

Line 6: Taking the address of `stack_free` transfers constraints projected from its declared signature (including the

where clause) onto a function pointer’s parameters and their components:  $\rho_{f.1} = 1, \rho_{f.1.e} = 0$ .

Line 7: The specification on `stack_free_all` says that the first argument takes an owning pointer and makes it non-owning. In addition, ownership of the elements in the first argument and their components must match that of the arguments of the function pointer:  $\rho_{ss_5} = \rho_{ss_7} + 1, \rho_{ss.e} = \rho_{f.1}, \rho_{ss.e.e} = \rho_{f.1.e}$ .

Line 8: A deallocator takes an owning pointer and makes it non-owning:  $\rho_{p_3} = \rho_{p_8} + 1$ .

Line 9: Variables are non-owning when they exit their scope:  $\rho_{s_5} = 0, \rho_{ss_7} = 0, \rho_{p_8} = 0$ .

The ownership constraints that are generated have a simple solution that satisfies all of them when  $\rho_{p_2}$  transfers ownership to  $\rho_{p_3}$  instead of into  $\rho_{s.e}$ , the elements of the stack pointed to by `s`. The solution is:

$$\begin{aligned} \rho_{ss_4} &= \rho_{ss_5} = 1 & \rho_{ss_7} &= 0 \\ \rho_{s_1} &= \rho_{s_3} = \rho_{ss.e} = \rho_{f.1} = 1 & \rho_{s_5} &= 0 \\ \rho_{p_2} &= \rho_{p_3} = 1 & \rho_{s.e} &= \rho_{ss.e.e} = \rho_{f.1.e} = 0 & \rho_{p_8} &= 0 \end{aligned}$$

Because there exists a solution that satisfies all of the constraints, we know that there are no leaks of objects and no object is deleted more than once.

### 6.3 Reporting Constraint Violations

A single error in the program can lead to multiple related constraint violations. In [21], each constraint violation was reported as a separate warning, resulting in a multiple distinct warnings for the same problem. In order to reduce the number of warnings that need to be examined, we compute partitions of connected components of the original set of constraints within a procedure. All constraint violations within a partition are grouped together in a single warning.

```
void multi_violation() {
  void *a = malloc();           (1)
  if (c0) {
    free(a);                    (2)
  } else if (c1) {
    use(a); // no deallocation (3)
  } else {
    use(a); // no deallocation (4)
  }
}
```

Figure 6: Multiple constraint violations

In the example code in Figure 6, `a` must be an owning pointer after (1) and before (2). Our analysis computes that it must also be owning before (3) and (4). If `use` does not deallocate or take ownership of its argument, the constraints that require `a` to be an owning pointer before, but a non-owning pointer after statements (3) and (4) will be violated. Because the ownership of `a` before the `if` statements is equivalent across each branch of control flow, these violated constraints will be combined into a single warning.

## 7. EXPERIMENTAL RESULTS

We have incorporated our container modeling and polymorphic ownership into the original Clouseau system, implemented in the SUIF2 [28] compiler infrastructure. Our

tool works for both C and C++ programs. We report on experiments used to evaluate the applicability and practicality of our model across five realistically large C and C++ packages including over one million lines of code.

### 7.1 Evaluation of the Model for C and C++ Programs

We applied our analysis to over a million lines of code in the following C and C++ packages: `gentoo`, a windowed application compiled with the `gtk+` and `glib` libraries; `OpenSSL`, a widely used cryptographic library; `MPlayer`, a multi-media player; an internal development version of the SUIF2 compiler; and a development version of a proprietary commercial hardware description compiler denoted here as `hwc`. These programs are all in active development and all except `OpenSSL` contained some leaks in a the dynamic execution of a test program.

Table 1 summarizes for each package the number of lines of code reported by SLOCCount [32], the analysis time, the number of polymorphic containers identified, the number of procedure specifications written for these containers, the number of specifications written for other routines, the number of leaks, double deletions and total errors identified with static analysis.

Most of the polymorphic containers and implementation routines that needed specification were identified quickly and automatically. For the SUIF2 compiler, we used our first-hand knowledge to identify the template classes and functions that implemented polymorphic containers. For the rest of the programs, we used an initial run of Clouseau without container specifications to identify potential container implementation routines as described in Section 2.4. All but one of the polymorphic containers in these programs were identified with this initial analysis.

Writing specifications for our experiment was relatively simple and fast. It required only understanding potential ownership transfers between contained elements and procedure parameters. This took no more than a few minutes for each function. An ownership specification can be reused throughout the life of the software. In `gentoo` for example, the specifications generated can benefit any application that uses the `glib` or `gtk+` libraries.

All of the applications except `MPlayer` use polymorphic containers that follow the ownership model. Providing a polymorphic specification for these containers removes some warnings and increases the precision of some low-level warnings over the basic un-annotated model.

The specification language is useful for more than just polymorphic containers. We report the number of other specified interfaces in the “Other Procedures” column of Figure 1. These procedures included some monomorphic containers, and a few reference counting implementations that could be effectively modeled with ownership.

This experiment demonstrates that enforcing the ownership model with Clouseau is effective at identifying memory problems for all of these applications. We found a total of 592 errors, including 10 double deletions. These are more serious than memory leaks since they can easily cause program failure and can be potentially exploited for remote system attacks [8].

Application	Lines of Code	Analysis Time	Polymorphic Containers	Polymorphic Procedures	Other Procedures	Leaks	Double Deletes	Total Errors
gentoo	34K	2m 20s	4	51	0	31	0	31
OpenSSL	187K	6m 14s	1	6	50	102	3	105
MPlayer	412K	8m 01s	0	0	10	70	7	77
SUIF2	328K	19m 58s	4	42	9	90	0	90
hwc	125K	4m 09s	4	18	7	324	1	325
Total	1086K	40m 42s	13	117	76	617	11	628

Table 1: Applications, Polymorphic Containers and Procedure Specification Counts

## 7.2 Error Identification

Application	Warnings	FP (%)	Esc
gentoo	48	41 (85%)	130
OpenSSL	351	246 (70%)	524
MPlayer	121	52 (43%)	739
SUIF2	281	226 (80%)	274
hwc	350	177 (51%)	1306
Total	1151	742 (64%)	2973

Table 2: False Positive Rates of the Final Specification on the Original Source

Identifying errors and fixing them with our tool is an iterative process, like the process of syntax checking a program. Fixing an error in the source code can eliminate warnings, but it can also lead to the identification of other errors and lead to opportunities for further specification of container or procedure interfaces. In the original experiment, we fixed errors, added specifications and re-ran the analysis until we could find no more errors. Because our analysis is fast, we can re-analyze a program when a source file or specification changes in a matter of minutes. Post-processing allows a user to ignore warnings that have been previously identified as spurious. The program is re-analyzed whenever a previously identified problem is found to propagate spurious warnings to other parts of the program.

To separate the false-positive rates for our analysis from the iterative process described above, we perform an experiment that analyzes one program from each package with the final specification. We reviewed all of the high-level warnings and identified the warnings that did not indicate one or more errors in the program. Table 2 gives the number of high-level warnings and the number and percentage of these warnings that are false-positives. For completeness, the last column presents the number of warnings about ownership that escapes our model. These are warnings required for soundness and are not reviewed for errors.

Warnings point to locations in the code where the ownership model is violated. In many cases these directly identify problems. Some programming idioms will cause expected warnings from our analysis. For example, our analysis will always generate warnings when a boolean condition, rather than the null-ness of a pointer is used to track ownership. Experience with the code can help find errors associated with an otherwise expected warning. Reviewing the partitioned warnings in `OpenSSL` helped us find an additional 36 errors beyond those reported in [20].

The false positive ratio among high-level warnings averages 64%. This is similar to that reported by previous flow-sensitive leak detection [21]. Hackett [18] and Xie [33] have more precise path-sensitive analyses that have identified bugs in `OpenSSL`. Xie et al. report a false positive ratio under 1% with a much longer single processor execution time. Their analysis does not attempt to identify double deletions found by ours. We identify leaks that they do not find, as well ones that had been fixed in the version they analyzed. They report a larger number of errors by analyzing all of the test programs, while we only analyzed one. Hackett uses a shape analysis to more precisely track memory locations from allocation sites, but the approach fails to finish for 3 allocation sites in a portion of `OpenSSL` and they report finding only 4 bugs from 13 warnings.

## 7.3 Effectiveness of Static Leak Detection

We use Valgrind [24] to evaluate our static memory leak detector by the volume of leaks identified. The version of Valgrind we used reports the volume of memory allocated but not deallocated or placed into persistent data structures. Valgrind can only report the stack frame where leaked memory was allocated and cannot pinpoint the source of an error, unlike our static tool. We measured the original volume of memory leaked by the program for a particular input set. We then measured memory leaks after fixing statically identified errors. Finally, we used a range of aggressive dynamic techniques to fix all of the program errors that we could reasonably identify that actually caused memory leaks.

The first column of Table 3 reports the volume of leaks in the original application. The next columns give the number of leaks and double deletes identified after our static analysis, the leak volume removed by fixing these problems, and the percentage of leak volume removed. The last set of columns present the number of leaks and double deletes identified with dynamic analysis techniques along with the leak volume removed with dynamic analysis.

In each case, static analysis allowed us to find leaks not identified with dynamic methods because they did not happen to occur in the program execution. In two packages, `MPlayer` and `hwc` we identified over 90% of the leak volume found by Valgrind on a single program execution. In `OpenSSL` our static analysis found 67 leaks and 2 potential double deletes even though the dynamic analysis was unable to identify any leaks. In `gentoo`, fixing the errors identified from the static analysis had only a minor impact on the volume of leaked memory. However, leaked memory is only a portion of the total memory used by the program. The errors we fixed reduced the total memory used by over 4.5 times the volume of leaked memory reported by Valgrind.

Application	Original Leak Volume	Clouseau					Dynamic Analysis			
		Errors Found			Leak Volume Removed	Pct	Errors Found			Leak Volume Removed
		Leaks	Double Deletes	Total			Leaks	Double Deletes	Total	
gentoo	28K	31	0	31	1K *	3.6%	8	0	8	1K
OpenSSL	0	102	3	105	0	0	0	0	0	0
MPlayer	1,591K	70	7	77	1,581K	99.4%	15	0	15	1,589K
SUIF2	121K	90	0	90	23K	19.0%	59	3	62	121K
hwc	53,234K	324	1	325	49,039K	92.1%	171	0	171	53,230K
Total		617	11	628			253	3	256	

**Table 3: Number of Errors and Leak Volumes Identified with Static and Dynamic Analysis. \* An Additional 132K of Weakly Referenced Memory was Removed from gentoo with Static Analysis**

In SUIF2, our analysis did not perform as well. It was only able to reduce the leak volume by about 19%. However, the procedure signatures computed by the static analysis were found to be useful during the dynamic leak detection process.

Static analysis is more effective than dynamic instrumentation in finding leaks throughout a program because it is not restricted to instrumented executions. It is also effective at identifying the source of leaks. Dynamic instrumentation is more effective at prioritizing leaks by volume on a particular execution. However, we found it difficult in many cases with dynamic leak detection to identify the programming errors associated with dynamic leak warnings.

Attempting to use dynamic methods to remove all of the leaks in a program, especially ones with reference counting and user-defined allocators was very time consuming. It often required temporary program modifications, debugging with hardware watchpoints, and manual review of the program. In cases where we resorted to manual review of the program, the ownership signatures produced by the static analysis helped tremendously in reducing the amount of inspected code.

## 7.4 Detailed Study of a Leaky Program

Since polymorphic ownership is most important in `hwc` and it is the leakiest of the programs that we studied, we looked more closely at this program. We ran our analysis first with the limited ownership model in [21], then with the ownership model presented here.

Techniques	Errors Found	Remaining Volume	Accounted for:	
			Volume	%
Monomorphic	195	32.5 MB	20.7 MB	38.9
Polymorphic	130	4.2 MB	28.3 MB	53.2
Static Total	325	4.2 MB	49.0 MB	92.1
Dynamic	171	~ 0 MB	53.2 MB	~ 100.0

**Table 4: Number of Errors and Eliminated Dynamic Volume Out of the 53.2 MB Leaked**

The first row of data in Table 4 reports the result of applying the ownership model without container support to the program: 508 high-level warnings, 38% or 195 of which were errors. It also reports a large number of low-level warnings which were not examined for memory leaks. Fixing the high-level warnings reported from the automatic analysis

eliminated about 38.9% of the leak volume found dynamically.

The errors identified by the addition of container interfaces after monomorphic ownership more than doubled the leak volume identified statically; going from 20.7 MB of leaks to 49.0 MB and 92.1% of the dynamic leak volume. The container-related errors were remarkably accurate as well. Of the 72 warnings related to containers, 70 of them were errors, achieving a correct reporting rate of 97%.

## 8. RELATED WORK

This work builds on previous work in static leak detection, instantiation-based type systems, linear types, object ownership models and capability-based type systems. In particular, our model is directly derived from that used in Clouseau [21]. Every object has an owner at each program point, ownership can be transferred, and arbitrary aliases are allowed. These properties help find memory leaks, but do not eliminate references to dangling pointers. This paper shows how ownership can be extended to handle polymorphic containers to help find the majority of memory leaks in practice and pinpoint the cause of the errors.

**The Ownership Model.** Our ownership model [21] is geared towards memory management and is different from the concept of ownership type [7, 25]. Ownership types have been designed to enforce object encapsulation, and have been used to prevent data races and deadlocks [2]. Extensions make ownership types more flexible [1, 2, 5, 6].

Our model and linear types use the concept of passing ownership between variables. However, unlike linear types, assignments are not required to transfer ownership. Modeling the optional transfer with a constraint allows our model to be applied to real code. Strict linear types require that the right-hand-side pointer in an assignment be nullified. Extensions have been proposed to make linear types more flexible [3, 30]. Alias types have been proposed to allow limited forms of aliasing by specifying the expected data memory shape properties [27, 31]. Linear types have been applied to track resource usage [12] and verify the correctness of region-based memory management [9, 14, 15, 16, 17, 29].

Recent work with three-valued logic for automatic precise pointer shape analysis can prove the absence of memory leaks for some complex data structures [10, 26] though scalability has not been demonstrated. Hackett [18] presents a more scalable demand-driven analysis that tracks heap objects and can precisely analyze some acyclic list imple-

mentations. They can also statically detect some memory leaks, dangling dereferences, and multiple deletions of objects. Xie [33] use boolean satisfiability to add path-sensitivity to static memory leak detection. Their results demonstrate very good false-positive ratios. These benefits come at the expense of significantly longer analysis times.

**Experimental Results.** Fully automatic, though unsound, tools that have been found to be effective in finding program errors in large programs include PREFIX [4] and Metal [11, 19]. These systems do not have a notion of object invariants, and to our knowledge do not recognize the importance of containers for leak detection. Auditing the low-level warnings resulting from the sound system in our previous work heightened our interest in containers.

## 9. CONCLUSIONS

We created a formalization and a tool for static memory leak detection based on a model of memory management which handles polymorphic containers. In a study of five programs with one million lines of code, we demonstrate the potential to greatly increase the effectiveness of memory leak detection. The technique significantly reduced dynamic memory leak volumes in 3 of the 4 programs with dynamic leaks. We identified 628 program errors including 11 double deletions and found memory leaks in all of the programs.

Our ownership inference algorithm has three other uses. First, because our analysis is sound, all procedures that lose ownership are identified by warnings helping to identify those container implementation routines that can benefit from user-supplied specifications. Second, the analysis propagates all the constraints flow- and context-sensitively to find all the violations of the interfaces automatically giving highly accurate reports on errors related to the usage of polymorphic containers. Third, the automatically derived specifications are helpful to programmers in writing new code and debugging dynamic leaks.

## 10. REFERENCES

- [1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA 2002*, pages 311–330, November 2002.
- [2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*, pages 211–230, November 2002.
- [3] J. Boyland. Alias burying: Unique variables without destructive reads. *Software-Practice and Experience*, 31(6):533–553, May 2001.
- [4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.
- [5] D. Clarke. An object calculus with ownership and containment. In *FOOL 2001*, January 2001.
- [6] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA 2002*, pages 292–310, November 2002.
- [7] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA 98*, pages 48–64, October 1998.
- [8] M. Corporation. CAN-2004-0416. Common Vulnerabilities and Exposures (CVE) (cve.mitre.org), 2004.
- [9] K. Cray, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *POPL 99*, pages 262–272, January 1999.
- [10] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Proceedings of the Static Analysis Symposium*, pages 115–134, July 2000.
- [11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP 2001*, pages 57–72, October 2001.
- [12] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI 2002*, pages 13–24, June 2002.
- [13] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI 2000*, June 2000.
- [14] D. Gay and A. Aiken. Memory management with explicit regions. In *PLDI 98*, pages 313–323, May 1998.
- [15] D. Gay and A. Aiken. Language support for regions. In *PLDI 2001*, pages 70–80, May 2001.
- [16] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 28–38, August 1986.
- [17] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI 2002*, pages 282–293, June 2002.
- [18] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. pages 310–323, January 2005.
- [19] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI 2002*, pages 69–82, June 2002.
- [20] D. L. Heine. *Static Memory Leak Detection*. PhD thesis, Department of Electrical Engineering, Stanford University, December 2004.
- [21] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI 2003*, pages 168–181, June 2003.
- [22] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
- [23] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83–101, January 1993.
- [24] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Third Workshop on Runtime Verification (RV’03)*, July 2003.
- [25] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP 98*, pages 158–185, July 1998.
- [26] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.
- [27] F. Smith, D. Walker, and G. Morrisett. Alias types. In *ESOP 2000*, pages 366–381, April 2000.
- [28] SUIF Group. The SUIF2 compiler system. <http://suif.stanford.edu/suif/suif2/>.
- [29] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.
- [30] P. Wadler. Linear types can change the world. In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, April 1990.
- [31] D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177–206, 2001.
- [32] D. A. Wheeler. More than a gigabuck: Estimating gnu/linux’s size, June 2001. <http://www.dwheeler.com/sloc>.
- [33] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ESEC/FSE-13*, Sept. 2005.