A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector

David L. Heine and Monica S. Lam

Computer Systems Laboratory
Stanford University
{dlheine, lam}@stanford.edu

ABSTRACT

This paper presents a static analysis tool that can automatically find memory leaks and deletions of dangling pointers in large C and C++ applications.

We have developed a type system to formalize a practical ownership model of memory management. In this model, every object is pointed to by one and only one *owning* pointer, which holds the exclusive right and obligation to either delete the object or to transfer the right to another owning pointer. In addition, a pointer-typed class member field is required to either always or never own its pointee at public method boundaries. Programs satisfying this model do not leak memory or delete the same object more than once.

We have also developed a flow-sensitive and context-sensitive algorithm to automatically infer the likely ownership interfaces of methods in a program. It identifies statements inconsistent with the model as sources of potential leaks or double deletes. The algorithm is sound with respect to a large subset of the C and C++ language in that it will report all possible errors. It is also practical and useful as it identifies those warnings likely to correspond to errors and helps the user understand the reported errors by showing them the assumed method interfaces.

Our techniques are validated with an implementation of a tool we call Clouseau. We applied Clouseau to a suite of applications: two web servers, a chat client, secure shell tools, executable object manipulation tools, and a compiler. The tool found a total of 134 serious memory errors in these applications. The tool analyzes over 50K lines of C++ code in about 9 minutes on a 2 GHz Pentium 4 machine and over 70K lines of C code in just over a minute.

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*

General Terms

Algorithms, Experimentation, Languages, Verification.

Keywords

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

1. INTRODUCTION

Many existing programs are written in C and C++ where memory needs to be manually managed. Leaking memory and deleting an object multiple times are some of the most common and hard-to-find errors in these programs. Memory leaks can cause memory-intensive and long-running programs, such as server codes, to fail. Deleting an object that has already been deleted can cause memory corruption.

1.1 Existing Tools

Dynamic tools like Purify are commonly used for detecting memory management errors in programs[20]. Purify instruments a program and reports all allocated memory at program exit that is not pointed to by any pointers. Unfortunately, it can only find leaks that occur during instrumented executions and incurs a nontrivial run-time overhead. Moreover, Purify can only identify the allocation sites of leaked memory; users are still left with the difficult task of finding the statements that cause the leak.

A number of automatic static analysis tools, such as PREfix[6] and Metal[12, 19], have also been developed that help find memory leaks in programs. These tools aim to identify those cases when a program loses all references to an object without having deleted the object. However, without global pointer alias analysis, these tools cannot track pointers stored in recursive data structures and container objects. Thus, they can only find memory leaks that occur close to allocation sites, missing many errors that account for important leaks in real programs.

Techniques that require user participation have also been proposed. *Linear types* allow only one reference to each dynamically allocated object[26]. This restriction is enforced by nullifying the contents of a linear-typed variable whenever it is read. That is, whenever a pointer assignment, q = p, is executed, the value of p is nullified and q becomes the sole reference of the object pointed to by p. Memory management is simple with linear types; deleting an object whenever its unique reference is destroyed guarantees no memory leaks, no double deletes and no dangling references. Unfortunately, it is difficult to program with the semantics of linear types.

The `auto_ptr` template class in the C++ standard library is an attempt at a practical solution. An `auto_ptr` *owns* the object it points to; whenever an `auto_ptr` is destroyed, the object it points to is automatically deleted. An `auto_ptr` is nullified whenever it is copied; however, unlike linear types, multiple references are allowed. Similar concepts are adopted by LCLint[13], which require programmers to decorate their pointer variables with *only* and *shared* keywords. The burden imposed on the programmer and the lack of support for polymorphic ownerships in functions render these techniques unappealing.

1.2 Contributions

We have developed a fully automatic technique that can find memory leaks and multiple deletes in large C and C++ programs. It is powerful enough to find leaks of objects whose references have been stored in recursive data structures in C++ programs. The main contributions of this paper include the formalization of a practical memory management scheme based on object ownership, a fully automatic ownership inference algorithm, and validation of the ideas via an implementation and an experiment.

1.2.1 A Practical Object Ownership Model

One way to ensure that no memory is leaked is to check that an object is deleted whenever its last reference is obliterated. Enforcing such a property is difficult as it requires, among other things, tracking all pointers to recursive data structures carefully. In fact, this is too hard even for programmers who, when dealing with objects that have multiple aliases, often resort to dynamic reference-counting schemes. However, dynamic reference counting can be expensive and cannot handle circular data structures.

In practice, programmers often adopt simple programming conventions to help them manage memory. One such convention is *object ownership*, which associates with each object an *owning pointer*, or *owner*, at all times. An owning pointer holds the *exclusive right and obligation* to either delete the object it points to or to transfer the right to another owning pointer. Upon deleting the object it points to, an owning pointer becomes *non-owning*. One or more *non-owning* pointers may point to the same object at any time. Note that this notion of ownership is different from the concept of ownership for encapsulation[7, 9, 21], which assumes that all accesses to an object must go through an owning pointer.

This model eliminates the possibility of deleting an object more than once and guarantees the deletion of all objects without references. However, like reference counting, this model may leak objects whose owning pointers form a cycle. This is rare in practice because most programs are designed with acyclic ownership structures.

Well-designed object-oriented programs often have simple easy-to-use external method interfaces. One common idiom is to require that pointer member fields in an object either *always* or *never* own their pointees at public method boundaries. Correspondingly, the destructor method of an object contains code to delete all and only objects pointed to by owning member fields. By adopting this assumption, our tool can reason about object ownership with the same relative ease afforded to programmers.

We have developed a formal type system for a small language to capture the object ownership model as described above. We have also expanded this formal model to handle most of the safe features of the full C and C++ languages. Our ownership model is applicable to real programs because:

1. Unlike linear types, multiple *non-owning* references are allowed to point to the same object.

2. Ownership can be transferred. Each object has an owner immediately upon creation, and the ownership may be transferred from owner to owner until the object is deleted.

3. Normal assignment and parameter passing semantics are supported by allowing ownership to be optionally transferred. In statement `u = v`, if `v` owns its object before the statement, it can either retain ownership, or transfer ownership to `u` and become a non-owning pointer itself after the assignment.

4. Object member fields are required to have the same ownership only at public method boundaries. More specifically, this invariant is required to hold only at method invocations where the receiving object may not be the same as the sending object.

5. Polymorphic method interfaces are supported. The ownership type signatures of methods are represented by constraints on the ownerships between input and output parameters. The same method may be invoked using parameters with different ownerships as long as the constraints are satisfied.

Note that we are not advocating the exclusive use of an ownership model for all objects. It is sometimes appropriate to manage some objects in a program using other techniques, such as reference counting and region-based management. Our system does not require that memory be managed solely with the ownership model.

1.2.2 Automatic Ownership Inference

Our algorithm automatically infers the ownership signatures for all pointer member fields and method interfaces directly from the code and can be applied to existing C and C++ programs without any modification. The algorithm is sound, i.e. it reports all possible errors, for the large safe subset of C++ described in Section 3.5. The algorithm is also practical and useful for the following reasons. Being flow-sensitive and context-sensitive, the algorithm is powerful yet fast enough to run on real programs. It is designed to isolate the sources of errors and not let them propagate. It identifies those warnings most likely to correspond to errors. Lastly, it helps users understand the error reports by showing them the assumed method interfaces.

The object ownership model is designed to allow fast inference. Tracking properties of objects, such as the count in reference counting, requires tracking aliases of objects, which is expensive. Ownership is not a property associated with the objects being managed, but rather an abstract property associated with the pointer variables themselves. We place ownership inference constraints on pointer variables to ensure that ownership is conserved between assignments and parameter passing in procedures. Methods in C++ classes are allowed to transfer ownership into and out of their pointer member fields. By assuming that member fields are either always owning or never owning at public method boundaries, we can easily track member fields in the `this` object within methods in a class. Alias analysis is rendered unnecessary by not allowing ownership to be transferred to: pointer member fields outside of their class, static pointer variables, pointers to pointers or pointer arrays. This model, while simple, is sufficient to find many errors in real code.

A powerful flow-sensitive and context-sensitive ownership inference. The algorithm tracks ownerships flow-sensitively as they are transferred between local variables and context-sensitively as they are transferred between procedural parameters. Ownership is tracked with 0-1 valued variables where 1 means owning and

0 means non-owning. Ownership inference is modeled as solving limited forms of 0-1 linear inequalities. We represent method ownership types polymorphically and succinctly as 0-1 linear inequalities on the ownerships of input and output parameters. We also use a sparse representation akin to semi-pruned SSA form[5] to reduce the size of the constraints per method. While ownership constraints can be resolved with a general-purpose solver, we have developed a specialized ownership constraint solver for efficiency and to gain control over how inconsistencies are reported.

A sound and practical tool by prioritizing constraints. It can be difficult to deduce the actual cause of errors reported by polymorphic type systems. Especially because our model does not encompass all legal ways of managing memory, a sound algorithm can potentially generate many false positive warnings, which could render the tool ineffective. Our solution is to satisfy the more precise constraints first so as to minimize the propagation of errors associated with the less precise constraints. In addition, the warnings are ranked according to the precision of the constraints being violated. By concentrating on the high-ranked warnings, and with the help of the automatically computed method signatures, the user can find errors in the program with reasonable effort. The advantage of having a sound system is that researchers can browse through the low-ranked warnings to understand the system's weakness and uncover opportunities for further improvement.

1.2.3 Validation

We have implemented the algorithm presented in this paper as a program analysis pass in the SUIF2 compiler infrastructure. We applied the system, called Clouseau, to six large open-source software systems, which have a total of about 400,000 lines of code. Every one of the C++ programs in our experiment contains classes that conform to our object ownership model. Clouseau found 134 definite errors in the programs. It analyzed over 50K lines of C++ code in about 9 minutes on a 2 GHz Pentium 4 machine.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview of the design of our ownership model. Section 3 presents our formal ownership type system for a simple object-oriented programming language and describes how we extend the system to C and C++. We describe our type inference algorithm in Section 4 and present experimental results in Section 5. We describe related work in Section 6 and conclude in Section 7. Appendix A contains the static typing rules for our ownership type system.

2. OVERVIEW

This section presents a high-level overview and the rationale for our system, with the help of several examples. We describe how we create constraints to model ownership in a program and how we order the satisfaction of constraints to improve the quality of error reports.

2.1 Optional Ownership Transfer in Assignments

Pointers returned by allocation routines and pointers passed to deallocation routines are, by definition, owning pointers. Ownership of all other pointer variables needs to be inferred, as illustrated by the following simple example:

EXAMPLE 1. Assignment statements.

```
u = new int;     (1)
z = u;           (2)
delete z;        (3)
```

The object allocated in statement (1) is clearly not leaked because it is deleted in statement (3) through the alias z. Our system, however, does not track pointer aliases per se. Instead, it knows that u is an owner after the allocation function in statement (1) and that z is an owner before the delete operation in statement (3). Statement (2) represents a possible transfer of ownership from u to z. All the constraints can be satisfied by requiring that the ownership be transferred from u to z, and thus the program is leak-free. □

Notice that the analysis is necessarily flow-sensitive. The source and destination variables of an assignment statement may change ownership after the execution of the assignment. In addition, both forward and backward flow of information is necessary to resolve the ownership of variables, as illustrated by the example above. We model all statements as constraints such that if all the constraints extracted from a program can be satisfied then the program has no memory leaks. Our analysis is not path-sensitive; a variable must have the same ownership at the confluence of control flow paths.

2.2 Polymorphic Ownership Signatures

Our analysis uses an accurate fixpoint algorithm to compute polymorphic ownership interfaces for each method in the program. The analysis initially approximates the effect of each method by ignoring the methods it invokes. It then instantiates the approximation of the callee's constraints in the caller's context to create a better approximation. This step is repeated until the solution converges.

EXAMPLE 2. Recursive procedures.

```
int *id(int *a) {
   int *t = 0;
   if (pred()) {
      int *c = new int;
      t = id(c);
      delete t;
   };
   return(a);
}
```

The recursively defined id function returns the original input value, with the side effect of possibly creating and deleting a number of integers. Even though id is called with an owning input parameter and return value in the recursive cycle, our analysis correctly summarizes the function as requiring only that the input parameter and the return value have identical ownerships. □

2.3 Object Ownership

For C++ programs, member fields in objects are assumed to be either always owning or never owning at public method boundaries. Our analysis starts by examining a class's constructors and its destructor to determine the ownership signatures of its fields, i.e. whether the fields are owning or non-owning at public method boundaries. Because an object's member fields can only be modified by methods invoked on the object, our analysis only needs to track member fields of the this object intraprocedurally and across methods on the this object. At other method invocation boundaries, member fields take on the ownership specified by their signatures.

EXAMPLE 3. Object invariants.

```
class Container {
  Elem *e;
public:
  Container(Elem *elem) {
    e = elem;
  }
  void set_e(Elem *elem) {
    delete e;
    e = elem;
  }
  Elem *get_e() {
    return(e);
  }
  Elem *repl_e(Elem *elem) {
    Elem *tmp = e;
    e = elem;
    return(tmp);
  }
  ~Container() {
    delete e;
  }
}
```

Member field e can be easily recognized as an owning member of the Container class because the destructor ~Container deletes it. Once e has been identified as an owning member field, the analysis can infer the ownership of pointers passed to and returned from each method of the class. The constructor Container and set_e must be passed an owning pointer; the return value of get_e must not be owning; the argument and return of repl_e must both be owning. This example also illustrates a limitation of the model. In practice, many classes implement a polymorphic get_e method and a set_e that does not delete the original member field. The object invariant is temporarily violated by code that uses get_e first to retrieve an owning pointer then uses set_e to deposit an owning pointer in its place. Our analysis flags this idiom as a violation of the ownership model. □

2.4 Reporting Errors

It is important that a leak detection tool pinpoints the statements that leak memory. In our system, a memory leak shows up as an inconsistency among constraints collected from a number of statements. The tool should strive to identify which among these statements is likely to be erroneous. Our solution is to try to satisfy the more precise constraints first, classify those constraints found to be inconsistent as errors and leave them out of consideration in subsequent analysis. Furthermore, we rank all the identified errors according to the precision of the constraints being violated.

EXAMPLE 4. For the class defined in Example 3, suppose there is one illegal use of the Container class that deletes the element returned by get_e. Had the analysis considered the constraints generated by this usage first, it would conclude that member field e is not owning in the Container class. This would lead the analysis to generate errors for all correct uses of the class, including one for the ~Container destructor function. Our analysis avoids this problem because constraints from destructors and constructors are considered to be more precise and are thus satisfied first. □

Our system does not allow any writes of owning pointers to fields in C structures or arrays of pointers. Many violations of this restriction are expected in real programs. To avoid propagating such inaccuracy to many other statements in the program, our algorithm first analyzes the code assuming that indirect accesses can optionally hold owning pointers, then re-analyzes it with the stricter constraint to generate all the warnings necessary to make the system sound. More details on constraint ordering can be found in Section 4.3.

3. OBJECT OWNERSHIP CONSTRAINTS

This section presents a formal type system that models object ownership. We define a small object-oriented programming language with typed ownerships. This language associates 0-1 ownership variables with pointers to indicate whether they own their pointees. Every member field in a class has a field signature which indicates if it is owning or non-owning at public method invocations. Each method also has an ownership signature which specifies constraints on the ownership variables associated with parameters and results. Note that although this language is defined with declared field and method signature ownerships, our inference algorithm *automatically infers* these signatures from C and C++ programs.

Well-typed programs in our ownership type system can be shown, using an approach developed by Wright and Felleisen[28], to satisfy the following two properties:

PROPERTY 1. *There exists one and only one owning pointer to every object allocated but not deleted.*

PROPERTY 2. *A delete operation can only be applied to an owning pointer.*

Property 1 guarantees no memory leaks; objects not pointed to by any variables are always deleted. Together, both properties guarantee that objects can only be deleted once.

We first present the language in Section 3.1. Sections 3.2 and 3.3 describe the handling of intraprocedural constraints and interprocedural constraints, respectively. Section 3.4 describes how we handle null pointers. Finally, Section 3.5 describes extensions to cover most of the safe features in C and C++.

3.1 A Language with Typed Ownership

A program P consists of a set of classes $\mathrm{CL}(P)$; each class $c \in \mathrm{CL}(P)$ has an ordered set of member fields $\mathrm{F}(c)$, a constructor new_c, a destructor delete_c and a set of other methods $\mathrm{M}(c)$. Constructors and destructors can execute arbitrary code and call other methods. Constructors have a list of formal arguments and a return value; destructors have a single implicit this argument, and all other methods have an implicit this argument, formal arguments, and a return value.

A method body contains a scope statement, which can access the this variable, local variables and parameters. Field accesses are allowed only on the this object. The language has a number of assignment statements. Simple variables can be assigned from NULL values, variables, class fields, and method invocation results. Fields in a class can only be assigned from variables. The language also has a number of compound statements: composition, if and while.

This language has ownership variables associated with parameters for each method and member fields for each class. The ownership variables may be given two values $\kappa \in \{0, 1\}$, where $\kappa = 1$ means owning, $\kappa = 0$ means non-owning. Constraints on ownership are represented as 0-1 linear inequalities over ownership variables. These constraints form a semi-lattice. The top element $\top$ in this lattice is true, and the bottom element $\bot$ is false. The meet operator in the lattice corresponds to the conjunction of 0-1 linear constraints. $C_1 \leq C_2$ if and only if solutions satisfying C_1 also satisfy C_2.

Each member field f in class c has an associated *field ownership type* $\text{FOT}_c(f)$, $[\rho \ ; \ \rho{=}\kappa]$, which specifies that field f is associated with ownership variable ρ whose value is κ. Whenever a method is invoked on a receiver object other than `this`, fields in both the sender and receiver objects must obey the ownerships specified in their respective field ownership types.

Each method m in class c has an associated *method ownership type* $\text{MOT}_c(m)$, $[\vec{\alpha} \ ; \ C]$. $\vec{\alpha}$ specifies the ownership variables for `this`, the formal arguments, the class member fields at method entry and exit, and the return value. C specifies the set of linear constraints that must be satisfied by these ownership variables. The signatures for constructors include ownership variables for formal arguments, fields on method exit, and the return value. The signatures for destructors include ownership variables for `this` and fields on method entry.

Typing rules in our system ensure that once an object is created, its ownership is held by some pointer variable. This ownership may be transferred via assignments and parameter passing in method invocations or stored in an owning member field. When the object is finally deleted, the owning variable relinquishes ownership and all objects pointed to by owning member fields are also deleted. Typing judgments for statements in our system have the form

$$B; D; C \vdash s \Rightarrow D'; C'$$

which states that given a mapping from variables to their declared class type B, a mapping from variables to their ownership variables D, and a set of ownership constraints C, it is legal to execute the statement s, which results in a new ownership mapping D' and new constraints C'.

The static ownership typing rules are shown in Appendix A. Due to space constraints, we will discuss only one representative inference rule in detail to give readers a flavor for how the system is defined formally and focus on explaining the intuition behind the formulation.

3.2 Intraprocedural Ownership Constraints

Assignment Statements. Let us first consider an assignment statement $z = y$, where y and z are both simple variables. The inference rule for the simple assignment statement, rule STMT-ASGN, is given in Appendix A and reproduced here:

$$\frac{D \vdash y \colon \rho_y, z \colon \rho_z \qquad \rho_z', \rho_y' \ fresh \qquad C' = C \wedge \{\rho_z{=}0\} \wedge \{\rho_y{=}\rho_y' + \rho_z'\}}{B; D; C \vdash z = y \Rightarrow D[z{\mapsto}\rho_z', y{\mapsto}\rho_y']; C'}$$

The term $D[z{\mapsto}\rho_z']$ is the functional update of the map D that binds z to ρ_z' after removing any old binding.

The ownerships of both y and z can change after the statement. ρ_y and ρ_z represents the ownerships of y and z before the statement; ρ_y' and ρ_z' represent their ownerships after the statement. Since z is overwritten, it must not own its pointee before the statement ($\rho_z{=}0$), otherwise memory is leaked. The constraint $\rho_y{=}\rho_y' + \rho_z'$ guarantees the conservation of ownership required by Property 1. If y owns its pointee before the statement, it either retains ownership or transfers it to z after the statement. If y does not own its pointee, neither y nor z is an owner after the statement.

Within a method, fields of the `this` object are treated like local variables. We define two rules STMT-FLDASGN and STMT-FLDUSE to handle assignments to and from member fields of the `this` object. These are similar to their STMT-ASGN counterpart.

A NULL pointer can be treated as either owning or non-owning. In C++, it is legal to delete a NULL pointer. We have found this relaxed constraint useful for analyzing C as well. Thus, if the right hand side of an assignment statement is NULL, we only require that the overwritten variable before the assignment be non-owning as

given in the rule STMT-NULL. A new, unbound, ownership variable is generated to represent the ownership of the destination variable after the assignment.

Control flow statements. The rule for composition, STMT-COMP, is straightforward. Our treatment of control flow is path-insensitive. At join points, variables and fields must have the same ownership at the confluence of control flow paths, as reflected in rules STMT-IF and STMT-WHILE.

3.3 Interprocedural Constraints

Constructors (`new`) and destructors (`delete`) are special because the former are guaranteed to return owning pointers and the latter always expect owning `this` pointers. Thus, we have three separate inference rules, CONSTR-GOOD, DESTR-GOOD, and METHOD-GOOD, to specify when constructors, destructors, and all other methods are well-typed.

Local variables defined in the `scope` statement in each method must not carry ownership on entry and exit; return variables start out non-owning, but may carry ownership upon exit. The ownership constraints imposed by the method's scope statement on all externally visible variables are summarized by a method's ownership type (MOT). Variables that are externally visible include the implicit `this` parameter, input parameters, member fields at method entry and exit, and the return value.

To keep our type system simple, we have adopted the convention that all variables to be passed as input parameters are first assigned to fresh variables, which are then passed as parameters in their place. This implies that all owning actual parameters can be assumed to pass their ownerships to their formal counterparts. That is, formal input parameters may be owning or non-owning upon entry, but must not be owning on exit. This convention also avoids the complexity that arises when the same variable is passed to multiple arguments of a method.

We refer to method invocations where the sender and the receiver are known to be the same object as *internal*. All other method invocations are considered *external*. As indicated by the inference rule for internal method invocations STMT-INTCALL, each execution of $r = \texttt{this}.m(\ldots)$ involves instantiating the constraints in the declared method ownership type of m and replacing the formal ownership variables with fresh variables. Equations are set up between the ownerships of actual parameters and fields and their corresponding instantiated formal variables. The ownership of the implicit `this` argument may be optionally passed to the invoked method. Furthermore, the ownership of the return value is transferred to variable r in the caller, which must not hold ownership before the statement.

The inference rule for external method calls STMT-EXTCALL differs from the rule for internal calls in its treatment of member fields. Member fields in both the sender and receiver objects must match their respective declared field ownership types (FOT). The rules for constructors STMT-CONSTR and destructors STMT-DESTR are slight modifications of that for external calls due to their differing parameter and return signatures.

EXAMPLE 5. Shown in Figure 1(a) is a short C++ program with method ownership types included as comments. The method ownership type of n, $\text{MOT}_c(n)$, is:

$$[\alpha\,\alpha_s\,\alpha_t\,\alpha_f\,\alpha_f'\,\alpha_r' ; \alpha = 0 \wedge \alpha_t = 1 \wedge \alpha_f = 0 \wedge \alpha_s = \alpha_r' + \alpha_f'],$$

where α represents the ownership of `this` at entry, α_s and α_t are the ownership variables of the first and second arguments, α_f and α_f' are the ownership variables for the field `f` at method entry and exit, respectively, α_r' is the ownership variable of the returned result.

This signature says that the ownership of `this` is not passed into the method, the input parameter t must be an owning pointer, the member field f must be non-owning at method entry. In addition, this signature polymorphically captures the fact that the first argument s can be called with either an owning or a non-owning pointer and allows the ownership, if present, be transferred to either the member field f or the returned result.

It is easy to tell from reading the code in method m that all the three objects created are deleted and not leaked. The second invocation of method n is particularly interesting because the same object is passed as the first and second arguments to the method, a fact that our analysis captures as a constraint in the caller.

The ownership signature in method m has three ownership variables: α represents `this` at the entry of method m, α_f and α_f' represent field f at entry and exit. Statements where ownerships may change are labeled with the names of the freshly generated ownership variables. We use ρ_{a_i} to denote the ownership of variable a in method m generated after executing statement i. We use σ_b^j to denote the instantiated ownerships for variable b in the jth invocation of n. The extracted ownership constraints of method m are given in Figure 1(b).

To keep the example short, the constraints for CALL #1 and CALL #2 directly encode the choice of ownership transfer for each of its arguments. In addition, we have omitted the ownership variables associated with `this` and the initial ownership values of the local variables in m, all of which are constrained to be non-owning.

As shall be shown in Example 6, the seemingly complex constraints in Figure 1(b) have a simple structure and are satisfiable. Thus, there is no memory leak within the method. Projecting away all the externally invisible variables gives the constraints in the declared ownership type in Figure 1(a).

This example also illustrates how our system handles aliases. The signature of n does not depend on whether the input parameters are aliased. The alias between x and y is implicitly captured by an ownership constraint in the calling method on the pointer variables themselves, $\rho_{x_5} = \rho_{x_6} + \rho_{y_6}$. □

3.4 Handling Null Pointers

Programs often first check if a pointer is null before deleting the object:

```
if (p != NULL) delete p
```

This idiom is problematic for our path-insensitive system as described above because objects are deleted on one path but not the other. Our solution is to handle the predicates that test if a pointer is NULL by inserting an explicit NULL assignment to the pointer on the appropriate path. For example, the previous statement is translated to:

```
if (p != NULL)
    delete p
else
    p = NULL;
```

Since NULL is treated as both an owning and non-owning pointer, the analysis will correctly deduce that the pointer is no longer owning when the `then` and `else` branches merge. We also use an intraprocedural analysis to propagate NULL values in each method so that they can be more precisely represented as either owning or non-owning.

(a)
```
class c {
  int* f;
  // MOT_c(n) = [α α_s α_t α_f α_f' α_r';
  //    α=0 ∧ α_t=1 ∧ α_f=0 ∧ α_s=α_r' + α_f']
  int* n (int* s, int* t) {
    delete t;
    f = s;
    int* r = s;
    return r;
  }
  // MOT_c(m) = [α α_f α_f';   α=0 ∧ α_f=0 ∧ α_f'=0]
  void m () {
    int* u = new int; (1) // ρ_u1
    int* v = new int; (2) // ρ_v2
    int* w = n(u, v); (3) // ρ_u3 ρ_v3 ρ_f3 ρ_w3
                          // σ_s^1 σ_t^1 σ_f^1 σ_f^1' σ_r^1'
    delete w;         (4) // ρ_w4

    c* x = new int;   (5) // ρ_x5
    c* y = x;         (6) // ρ_y6 ρ_x6
    c* z = n(x, y);   (7) // ρ_x7 ρ_y7 ρ_f7 ρ_z7
                          // σ_s^2 σ_t^2 σ_f^2 σ_f^2' σ_r^2'
  } ...
}
```

(b)

$$\rho_{u_1} = 1 \wedge \rho_{v_2} = 1 \qquad\qquad\qquad\qquad [\text{stmt } 1, 2]$$
$$\rho_{u_1} = \rho_{u_3} + \sigma_s^1 \wedge \rho_{v_2} = \rho_{v_3} + \sigma_t^1$$
$$\quad \wedge\ \alpha_f = \sigma_f^1 \wedge \sigma_f^{1'} = \rho_{f_3} \wedge \sigma_r^{1'} = \rho_{w_3} \quad [\text{CALL \#1}]$$
$$\sigma_t^1 = 1 \wedge \sigma_f^1 = 0 \wedge \sigma_s^1 = \sigma_r^{1'} + \sigma_f^{1'} \quad [\text{MOT}_c(n)]$$
$$\rho_{w_3} = 1 \wedge \rho_{w_4} = 0 \qquad\qquad\qquad\quad [\text{stmt } 4]$$

$$\rho_{x_5} = 1 \wedge \rho_{x_5} = \rho_{x_6} + \rho_{y_6} \qquad\qquad [\text{stmt } 5, 6]$$
$$\rho_{x_6} = \rho_{x_7} + \sigma_s^2 \wedge \rho_{y_6} = \rho_{y_7} + \sigma_t^2$$
$$\quad \wedge\ \rho_{f_3} = \sigma_f^2 \wedge \sigma_f^{2'} = \rho_{f_7} \wedge \sigma_r^{2'} = \rho_{z_7} \quad [\text{CALL \#2}]$$
$$\sigma_t^2 = 1 \wedge \sigma_f^2 = 0 \wedge \sigma_s^2 = \sigma_r^{2'} + \sigma_f^{2'} \quad [\text{MOT}_c(n)]$$
$$\rho_{u_3} = 0 \wedge \rho_{v_3} = 0 \wedge \rho_{w_4} = 0 \wedge \rho_{x_7} = 0$$
$$\quad \wedge\ \rho_{y_7} = 0 \wedge \rho_{z_7} = 0 \wedge \rho_{f_7} = \alpha_f' \qquad [\text{EXIT}(m)]$$

Figure 1: (a) A short C++ program with method ownership types and (b) ownership constraints for method m.

3.5 Handling C and C++ Features

While our small language is object-oriented, it can also be used to model C by simply treating all functions as methods defined for one global `this` object. We have further extended the system to handle various C++ features including multiple inheritance with virtual dispatch, static functions, multiple constructors per class and templates. Use of these features in a program will not cause any unnecessary warnings in our system.

Our model currently requires atomic memory allocation and object construction, as well as atomic object destruction and deallocation. Breaking these invariants by using the C++ placement new syntax or making explicit calls to the destructor will generate warnings.

Our system does not yet model ownerships for the following C and C++ language features: aliases to the address of a pointer member field in a class, concurrent execution, exception handling, the use of unsafe typecasts, pointer arithmetic, and function pointers. The use of any of these features may prevent our system from identifying all potential leaks. Conversely, if none of these features are

used, our algorithm is sound and will report all the potential memory leaks.

4. OWNERSHIP INFERENCE

Our ownership inference algorithm automatically infers the "likely" ownership signatures of fields and methods. It identifies statements that are inconsistent with these signatures and warns of potential memory leaks and double deletes. Constraints are introduced in a carefully designed order so as to isolate the source of errors. If a constraint is found to be inconsistent with the already established constraints, it is discarded and its associated statement is labeled as an error. In the following, we first describe our specialized constraint solver used to determine if a set of constraints is consistent. Then we describe the interprocedural type inference algorithm. Finally we present the order in which constraints are considered.

4.1 Constraint Resolution

The constraints on ownership variables as shown in the inference rules in Appendix A are all 0-1 integer equalities. As introduced in Section 2.4 and further described in Section 4.3.3, 0-1 integer inequalities are also introduced by our algorithm to minimize error propagation. Ownership constraints are of the form:

$$\rho{=}0 \mid \rho{=}1 \mid \rho{=}\Sigma_i\rho_i \mid \rho \geq \Sigma_i\rho_i$$

We have developed a solver optimized to take advantage of the special properties held by ownership constraints. Our attempt to apply a general solver to this problem suggests that it is significantly slower than our special-purpose solver.

4.1.1 Ownership Graphs

Ownership constraints can be represented by a bipartite *ownership graph* $G = \langle N, V, E \rangle$, where N is a set of partitions of ownership variables, V is a set of *flow* nodes representing non-constant constraints, and E is a set of directed edges connecting nodes in N to nodes in V and vice versa. Each partition $n \in N$ may have a label, $L(n)$, which can either be 1 representing owning or 0 representing non-owning. Initially every ownership variable is placed in its own partition. Our solver gradually rewrites the ownership graph to create smaller, equivalent graphs, along with ownership partitioning functions $\pi : O \to N$, which map elements in the set of ownership variables O to their respective partitions.

Constant constraints are represented by labeling the partitions accordingly. Each non-constant constraint

$$\rho = \sum_i \rho_i \text{ or } \rho \geq \sum_i \rho_i$$

is represented by a flow node $v \in V$, an edge $(\pi(\rho), v)$ and edges $(v, \pi(\rho_i))$, for all i. Thus, a flow node has one incoming edge and possibly multiple outgoing edges. Flow nodes are so named because they have the following properties:

1. If the source of the incoming edge of a flow node is labeled 0, the outgoing edges must also have destinations labeled 0.

2. If the source of the incoming edge of a flow node is labeled 1, then at most one outgoing edge has a destination labeled 1. In the special case where the flow node represents an equality constraint, then exactly one outgoing edge has a destination labeled 1.

4.1.2 Consistency Checks

We now describe how our solver checks if a new constraint is consistent with a given set of consistent constraints, represented as an ownership graph $G = \langle N, V, E \rangle$ and a partitioning function π, and returns the combined set of constraints if no inconsistency is detected.

If the new constraint is a constant constraint, $\rho = \kappa$, $\kappa \in \{0, 1\}$, we assign κ to $L(\pi(\rho))$. If $\pi(\rho)$ has already been given a different label, the constraint is inconsistent. If the new constraint is a non-constant constraint, we represent the constraint by adding a new flow node and corresponding edges to G as described above. Next, we apply the rewrite rules below repeatedly to G until none is applicable. If any application of the rewrite rules requires assigning a labeled partition with a different value, the new constraint is found to be inconsistent.

Before describing the rewrite rules, let us define a few terms. We say that a node n *reaches* n' if there exists a path of edges in E that lead n to n'. To *unify* node n' with node n, we merge the partition of n' into n by assigning $L(n')$ to $L(n)$ and replacing flow edges (v, n') and (n', v) with edges (v, n) and (n, v), respectively. The rewrite rules we use are as follows:

[EDGEREMOVAL]. Remove a flow node and its edges if its predecessor and successor nodes are labeled.

[SINGLENODE]. If an equality flow node v has a single predecessor n and a single successor n', unify n and n'.

[ZEROIN]. If $L(n) = 0$, assign 0 to $L(n')$, for all n' reachable from n.

[ZEROOUT]. If $L(n) = 0$, eliminate all (v, n) edges in E.

[ONEOUT]. If $L(n) = 1$, assign 1 to $L(n')$ for all n' reaching n.

[MULTIPATH]. If a flow node reaches the same partition n via two distinct paths, assign 0 to $L(n)$.

[CYCLEELIMINATION]. Unify all partitions along a cyclic path and assign 0 to any partition reachable from the unified partition.

All but the last rule are self-explanatory. To understand the CYCLEELIMINATION rule, we consider two cases. If a partition in a cycle is labeled 1, it must pass ownership back to itself through the cycle, thus all other partitions along the cycle must also be labeled 1. Conversely, if a partition in a cycle is labeled 0, all other partitions must also necessarily be labeled 0. In either case, no ownership is passed downstream.

We refer to the ownership graph obtained by the rewrite procedure as the *canonical ownership graph*. It is easy to show that a canonical ownership graph consists of a set of unconnected directed acyclic graphs (DAGs), whose root partitions may be labeled 1 and the rest unlabeled. This graph is trivially satisfiable if one of the following is true:

1. None of the root nodes is labeled. Labeling all nodes 0 is a solution.

2. The labeled root nodes have no common descendant. A solution can be found by assigning 1 to an arbitrary descendant of a labeled root node and 0 to all others.

In the general case, if the above two simple tests fail, we can check the connected components of the DAG independently for satisfiability; these DAGs are found to be small in practice.

EXAMPLE 6. Figure 2 is the ownership graph derived from method m in Figure 1 with both calls to method n instantiated. Flow nodes are represented simply by the equal sign (=). Square
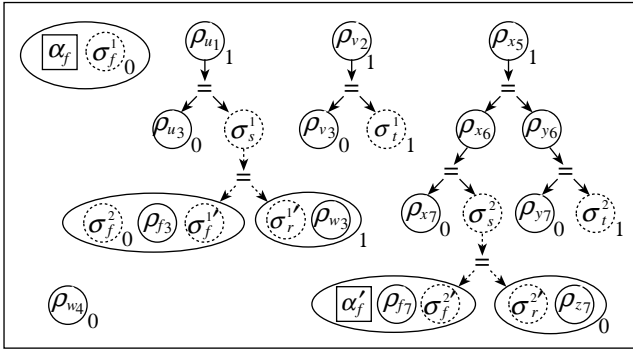
Figure 2: Ownership graph of method m in Figure 1 after instantiations of n.

nodes represent variables visible outside of m; circles represent variables internal to m. Each square and circle represents a partition, unless it is embedded in an oval, which represents the unification of the variables therein. Each partition node is labeled with its ownership assignment, if one is known. Edges in the ownership graph are represented by arrows. All nodes and edges instantiated from method n are represented by dotted lines. It is easy to see the solution from this graphical representation. The ownerships held by ρ_{u_1}, ρ_{v_2} and ρ_{x_5} all must flow down the right edge of each flow node because the rightmost descendant in each case is labeled 1. □

4.2 Interprocedural Analysis

Our algorithm is polymorphic, or fully context-sensitive, in that it finds the least constrained signature for each method and field ownership type in well-typed programs. The algorithm introduces no inaccuracy even in the presence of recursive cycles. This section describes how we find the method ownership types assuming that field ownership types have already been found. We will discuss how field ownerships are handled in Section 4.3.

The algorithm first creates a set of constraints representing the effect of each method, assuming that the called methods have ⊤ for constraints. It then iterates to find a better approximation until the solution stabilizes. Our algorithm precomputes strongly connected components from a program's call graph. Method evaluation is performed in reverse topological order on the components with iteration needed only within components. Whenever the ownership type of a method is changed, all the methods that invoke the changed method are re-evaluated.

To create the initial summary of each method, we collect the constraints for each statement in the method according to the inference rules presented in Appendix A, ignoring method invocations. To handle arbitrary control flow in C and C++ programs, we use an SSA-like algorithm to generate ownership variables in the method and unifications of ownership variables at the join points of control flow. Since ownerships may change for both the source and destination variables in an assignment statement, we have adapted the original SSA algorithm to create new ownership variables not only for the destination variables but also for the source variables. We prune the SSA-like form by removing all the unnecessary joins for variables whose live ranges are limited to one basic block[5]. We also apply intraprocedural dead-code elimination to further reduce unnecessary assignments in the code.

The solver defined in Section 4.1 is applied to the constraints collected to create a canonical ownership graph for each method.

Note that the system may be found to be inconsistent in the process and only consistent constraints are used to summarize a method.

In the iterative step, a method is visited only if the signature of at least one of its callees' has changed. For each callee with new constraint information, we first reduce its current ownership constraints to pure relations on formal parameters. We project away the internal partition nodes by connecting each input parameter to a fresh flow node, which is then connected to all output parameters reached by that input. This may create many more edges but limits the name space of the ownership partitions. These relations between formal parameters are then instantiated in the calling context by substituting actual parameters for the formal parameters. The solver is then applied to check the constraints for consistency and to simplify it.
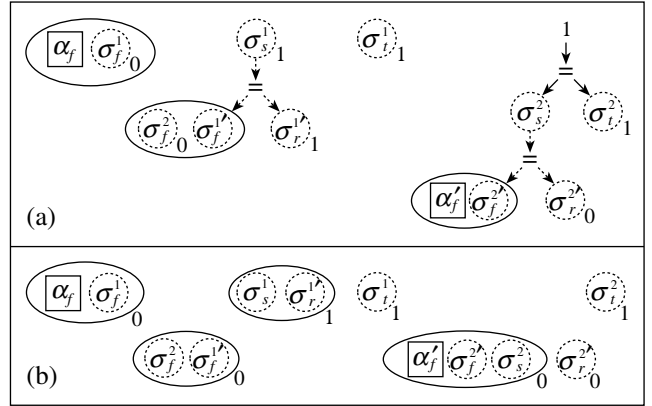
Figure 3: Ownership graph of method m in Example 5 after (a) intraprocedural analysis and (b) interprocedural analysis.

EXAMPLE 7. Returning to the example in Figure 1, our algorithm first creates an intraprocedural summary for methods n and m. It calls the solver to simplify the constraints internal to each method, ignoring the constraints of any callees, and projects away all internal variables other than parameters. The result for m after the first step is shown in Figure 3(a). All that remains in the ownership graph for m are the external parameters α_f and α_f' and the parameters passed into and out of its callee n. Note that the instantiated constraints, shown in dotted lines, are not visible to this first step; they are shown here only for reference.

The second step of the algorithm instantiates the constraints from n into the intraprocedural summary of m and applies the solver to the resulting graph, yielding the result shown in Figure 3(b). The final signature for m obtained by projecting away all internal variables (and including the ownership variable α for this, which is not included in the figure) is simply:

$$[\alpha\, \alpha_f\, \alpha_f'; \alpha = 0 \wedge \alpha_f = 0 \wedge \alpha_f' = 0].$$

□

One complication that arises in practice is that programs often invoke functions or methods for which no source code is available. These include system calls and calls to libraries that are compiled separately. We predefine method ownership types for a number of system calls like strdup, strcpy and memcpy. We also allow the user to give a specification for other routines. If no specification is provided, we assume that parameters to undefined functions are non-owning.

4.3 Constraint Ordering

Without a specification, it is impossible to tell which constraints among a set of inconsistent constraints are the erroneous ones. Thus we can only use heuristics to find the statements most likely to be incorrect. Misclassifying an erroneous statement as correct can result in misclassifying many correct statements as erroneous. This propagation of errors can result in many warnings, which require more user effort to diagnose.

Some statements are more precisely modeled than others. For example, allocations are precisely modeled because they will always return an owning pointer. On the other hand, the constraint that owning pointers cannot be stored indirectly is imprecise and is likely to be violated. Similarly, some methods are less likely to contain ownership errors than others. Destructors are more likely to be correct in handling member fields than methods that invoke them. Our approach to minimizing error propagation is to classify constraints according to their degree of precision and to try to satisfy the more precise constraints first. In addition, we rank warnings according to the precision of the constraints being violated.

4.3.1 Typing Fields Before Methods

We assume that the implementation of a class is a more reliable source for identifying an interface than code that uses the class. Therefore, our top-level algorithm has two steps: it first finds the ownership type of member fields in each class by considering only methods within the class; it then uses the field ownerships in a whole-program analysis to find the method ownership signatures.

To find class member field ownerships, we use an algorithm similar to the one described in Section 4.2. This step analyzes a class at a time. Because even constructors and destructors may invoke other methods, an interprocedural analysis is used. We model external invocations by simply assuming that constructors return owning pointers and destructors accept owning pointers as arguments and ignoring all other constraints. Member field ownerships are initialized to $\top$, meaning they can be either owning or non-owning. Destructors and constructors are analyzed before other methods. A member field is presumed to be owning if it is owning at the entry of the destructor or owning at the exit of any of the constructors. It is presumed to be non-owning otherwise.

4.3.2 Ranking Statements

The intraprocedural analysis orders the constraints so that the more precise constraints are satisfied first. Below is the ranking used, starting with the most precise:

1. Constant constraints including allocations, deletions, overwritten variables, and variables entering and exiting a scope.

2. Equality flow constraints with a single output, except those generated by uncommon control flow paths. These arise from join nodes and assignments without choice. Among flow nodes generated from joins, those with a lower fan-in are considered more precise.

3. Constraints generated by assignments with choice.

4. Flow constraints arising from joins on uncommon control flow paths like loop continues and breaks.

5. Constraints generated by loads and stores involving indirect pointer accesses.

4.3.3 Handling Inaccuracies in Methods

The order in which methods are analyzed is dictated by the call graph, since it is not possible to find the ownership signature of a method without the signature of its callees. This creates a problem because errors in summarizing a callee can propagate interprocedurally to all its callers.

There are three kinds of constraints that are of particular concern: the constraint that indirectly accessed pointers may not hold ownership, the constraint that undefined functions cannot have owning parameters and return values, and the constraint that fields in the sender object must honor their ownership type before and after external invocations. The last constraint is overly restrictive because most external calls do not access member fields in the sender object.

Our solution is to analyze the program with more relaxed versions of the above constraints. We relax the constraints to allow indirectly accessed pointers and parameters to undefined functions to optionally hold ownerships. These constraints are represented by 0-1 inequalities. A sender's member fields need not obey the field signatures across external method invocations. At the end of the analysis, we walk through each method, tighten the constraints, and report the rest of the violations.

5. EXPERIMENTAL RESULTS

Clouseau, the memory leak detector we implemented, incorporates all the techniques described in this paper. Clouseau is implemented as a program analysis pass in the SUIF2 compiler infrastructure. Besides finding potential errors in a given program, Clouseau also generates inferred ownership signatures of each procedure, class method and class member field. These signatures help users understand the warnings generated. Clouseau is sound within the safe subset of C and C++ presented in Section 3.5, meaning that it will report all possible errors in the program. Some of these warnings pertain to incorrect usage of the ownership model, which is the main objective of the tool. Some of these warnings are due to limitations of our analysis or the use of schemes other than object ownership in managing memory.

To evaluate the utility of the Clouseau system, we applied Clouseau to six large C and C++ applications. Our tool reported warnings on all the applications. We examined the warnings to determine if they correspond to errors in the program. We found errors in every package and 134 potentially serious memory leaks and double deletes were found in total.

5.1 The Application Programs

The application suite in this experiment consists of three C packages and three C++ packages. The C programs in the suite are widely used: GNU binutils, which are a set of object file generation and transformation tools, openssh, a set of secure shell clients and servers, and the apache web server. The C++ applications are under active development: licq, an internet chat client, Pi3Web, a web server written in C++, and the SUIF2 compiler's base package. The first two are available from `http://sourceforge.net` and SUIF2 is available at `http://suif.stanford.edu`.

Many of these packages contain a number of executable programs and libraries, as shown in Figure 4. The figure also includes other statistics on each package including the number of source files, the number of functions, and the lines of code (LOC), as measured by counting unique non-blank lines of preprocessed source code. In total, our tool analyzed about 390,000 lines of code in the experiment. We also report the size of the largest program in each package and the time ownership inference takes for that program.

The measurements were taken on a 2 GHz Pentium 4 machine with 2 GBytes of memory. These numbers do not include the time required by the front end, linker, and preprocessing transformations. It took 1.2 minutes to analyze 71K lines of C code and

Package	Exe	Lib	Files	Func	LOC	Largest Exe LOC	Largest Exe Time
binutils	14	4	196	2928	147K	71K	69
openssh	11	2	132	1040	38K	23K	13
apache	9	27	166	2047	66K	43K	29
licq	1	0	31	2673	28K	28K	240
Pi3Web	48	14	173	2050	40K	25K	85
SUIF2	12	30	203	8272	71K	55K	528
TOTAL	95	77	901	19010	390K		

Figure 4: Application characteristics: number of executables, libraries, files, functions, lines of code, lines of code in the largest executable and its ownership analysis time in seconds.

8.8 minutes to analyze 55K lines of C++ code. C++ program analysis is slower because of the extra intra-class analysis used to determine member field ownerships, the inclusion of member fields in the analysis, and the larger call graphs generated by class hierarchy analysis, We have not optimized our implementation, as the analysis was fast enough for experimentation.

5.2 The C Packages

Clouseau generated a total of 1529 warnings for the three C programs in the suite. The warnings are separated into three classes: violations of intraprocedural constraints, violations of interprocedural constraints, and *escaping* violations. Escaping violations refer to possible transfers of ownership to pointers stored in structures, arrays or indirectly accessed variables. While these warnings tell the users which data structures in the program may hold owning pointers, they leave the user with much of the burden of determining whether any of these pointers leak. Users are not expected to examine the escaping warnings, so we only examined the non-escaping warnings to find program errors.

Only 362 of the warnings are non-escaping; 82 are intraprocedural and 280 are interprocedural. We found altogether 85 errors, as shown in Figure 5. The error-to-warning ratio is 32% for intraprocedural violations and 21% for interprocedural violations. We found the method signatures generated by Clouseau helpful with our examination of the interprocedural warnings.

Package	Intraprocedural Reported	Intraprocedural Bugs	Interprocedural Reported	Interprocedural Bugs	Escapes
binutils	79	26	200	40	727
openssh	1	0	73	18	408
apache	2	0	7	1	32
Total	82	26	280	59	1167

Figure 5: Reported warnings and identified errors on C applications

Many of the errors in binutils and openssh are due to missing object deletions along abnormal control flow paths, such as early procedure returns or loop breaks. Some procedures return both pointers to newly allocated and statically allocated memory. Occasionally, deletes are missing in the normal execution of a loop. In openssh, almost all memory allocation and deletion routines are wrapped by other procedures, thus interprocedural analysis is a prerequisite to finding any leaks in the program. The apache web server illustrates an interesting scenario. Clouseau reported only 9 warnings, only one of which was found to be an error. Examination

of these warnings quickly revealed that apache manages its memory using a region-based scheme. While Clouseau does not understand region-based management, the few warnings generated succeed in helping users understand how the program manages memory.

5.3 The C++ Packages

Clouseau offers more functionality in finding leaks in C++ programs. By assuming that member fields are either owning or non-owning at public method boundaries, Clouseau can find errors associated with leaks stored in class member fields. For C++, Clouseau generates two more categories of warnings than those for C: violations of receivers' member field ownerships and violations of senders' member field ownerships in external invocations. The latter, as discussed in Section 4.3.3, are most likely caused by limitations in our model and not real errors in the program. The breakdown of each category of warnings and errors is shown in Figure 6.

We analyzed the receiver-field, intraprocedural and interprocedural violations to look for memory leaks and double deletes. We found two common suspicious practices, which we classify as minor errors. First, many classes with owning member fields do not have their own copy constructors and copy operators; the default implementations are incorrect because copying owning fields will create multiple owners to the same object. Even if copy constructors and copy operators are not used in the current code, they should be properly defined in case they are used in the future. Second, 578 of the 864 interprocedural warnings reported for SUIF2 are caused by leaks that occur just before the program finds an assertion violation and aborts. We have implemented a simple interprocedural analysis that can catch these cases and suppress the generation of such errors if desired. Counting the minor errors, 770 (69%) of the 1111 examined warnings lead to errors. Ignoring the minor errors and their warnings, 49 (13%) of the 390 examined warnings lead to errors.

This experiment shows that the object ownership model is used in all the three C++ applications. Ignoring the minor default copy constructors and copy operators problem, classes with owning member fields have no leaks in Pi3web and licq. In the SUIF2 compiler, however, some classes leak owning members because either the class destructor does not delete them or the default destructor is used. We also identified two serious errors where an object can be deleted twice. Double deletes are even more significant than memory leaks because they can cause memory corruption and undefined behavior.

5.4 Discussion

Our experience in working with real-life applications leads to a number of interesting observations. First, the field and method signatures generated by Clouseau help explain to the programmer the cause of the warnings. It has been found that good explanations are required for effective error detection tools[6]. Our generated method interfaces allow users to reason about the methods one at a time. Even when the generated method interfaces are erroneous, programmers can often easily detect such errors.

Second, automatic tools are not affected by some of the problems that can mislead programmers. One error we found in binutils was caused by a misnomer. The function bfd_alloc does not return ownership of newly allocated objects despite its name; giving no credence to function names, Clouseau easily derived this fact from the implementation. As another example, some of the leaks in licq are generated by macros that expand to include early loop exits that cause memory leaks. It is difficult for a programmer, without looking at the preprocessed code, to find these leaks. These examples

| | Receiver-Field | | | Intraprocedural | | Interprocedural | | | | Sender-Field |
Package	Reported	Major	Minor	Reported	Major	Reported	Major	Minor	Escapes	
Pi3Web	38	0	33	10	0	46	4	0	134	36
licq	42	0	40	33	14	114	16	0	231	622
SUIF2	91	8	70	33	5	704	2	578	523	886
Total	171	8	143	76	19	864	22	578	888	1544

Figure 6: Reported warnings on C++ applications, with identified major and minor errors

suggest that tools can find new program errors even in well-audited code.

Third, without using or needing a specification, Clouseau can only find inconsistencies in programs and cannot say, for sure, which statements in the program are wrong. Inconsistencies are often interesting to programmers, even if they do not correspond to erroneous memory usage. In some code in binutils, a programmer called the wrong function by mistake. The function happened to have an ownership signature different from the intended function. By examining the ownership inconsistency reported by Clouseau, we were able to discover an otherwise hard-to-find error. Some of the inconsistencies are due to unusual programming styles. Clouseau found a suspicious case where a pointer to the middle of an object is stored as an owning pointer. The pointer happens to be suitably decremented to point to the beginning of the object before it is freed. Identification of such unconventional pointer usage might be of interest to a programmer.

Fourth, detecting program errors is but the first step towards fixing the errors. Fixing memory leaks, especially when errors are caused by poorly designed interfaces, can be hard. A few of the leaks we found appear to have been discovered before, based on comments in the source files near the reported problems. They have not been fixed presumably because of the complexity involved. In fact, after we reported one such interface problem in binutils, a developer introduced another bug in an attempt to fix the error. Incidentally this bug would have been caught by Clouseau had it been run after applying the fix.

While Clouseau successfully helped identify a number of errors in large complex programs, our experiment also revealed some weaknesses in our current system and suggests several interesting avenues for future work. It would be useful and possible to reduce the number of false warnings among the receiver-field, intraprocedural and interprocedural violations. We found that predicates are often used to indicate whether pointers own their objects, thus adding path sensitivity will reduce the number of false warnings. There are many receiver-field violations due to the idiom of using a combination of `get` and `set` to replace an owning member field. Relaxing the ownership model and improving the analysis to handle such cases would be useful.

More importantly, we need to help programmers find leaks among the escaping violations. We found that C and C++ programs often pass pointers to pointers as parameters to simulate a pass-by-reference semantics. Better handling of such pointers would be useful. We also need to improve the handling of objects in containers. Our investigation suggests that containers can be handled by having the user specify the relationships between containers and their elements in the code and augmenting the static analysis described in this paper. Details of this work are the subject of another paper.

6. RELATED WORK

This research builds upon previous work on linear types, object ownership models and capability-based type systems. The design of our system is driven mainly by our goal to build a tool that can be applied to the large existing C and C++ code bases.

Ownership model. Our work is based on a notion of ownership that has been adopted by practitioners and shares many similarities with the concepts embodied in `auto_ptr` and LCLint[13]. Owning pointers help find memory leaks, but do not eliminate references to dangling pointers. Our model adds optional ownership transfer in assignment, allows arbitrary aliases, and includes an object ownership invariant at public method boundaries. We have formalized the ownership model and developed an algorithm that can automatically infer polymorphic ownership signatures of the methods and fields in C and C++ programs.

Clarke et al. proposed a stricter model of ownership known as ownership type[9, 21]. The basic notion is that an object may own another subobject. An object is always owned by the same object; furthermore, ownership types enforce object encapsulation, that is, objects can only be referenced through owner pointers. This notion of encapsulation restricts access to the subobjects, thus allowing one to reason about a subobject by only considering methods in the object. This has been used in preventing data races and deadlocks[2]. Various extensions have been proposed to allow restricted forms of access to objects without going through the owner[2, 7, 8]. AliasJava uses a more flexible ownership type system[1]. While it is still not possible to change the ownership of an object, an owner can grant permission to access an object to another object, and aliases can be temporarily created to owned objects. More recent work has enabled the expression of parameterized containers and their iterators while enforcing encapsulation properties[3].

Since ownership can be passed around in our system, our work also bears some similarities with linear types. Because we allow non-owning pointers, assignments may or may not transfer ownership; the option is captured by a constraint, and the system is type-safe as long as the constraints derived are satisfiable. Strict linear types require the right-hand-side pointer be nullified. Language extensions have been proposed to allow read-only aliases[26]. Boyland's alias burying allows aliases but the aliases are all nullified when one of them is read[4]. He proposed an inference algorithm that requires annotations be placed on procedural interfaces. Alias types have been proposed to allow limited forms of aliasing by specifying the expected data memory shape properties[24, 27]. Linear types have been applied to track resource usage[14] and verify the correctness of region-based memory management[10, 15, 16, 17, 18, 25]. Finally, Dor et al. propose checking for memory leaks using a sophisticated pointer shape analysis[11]; unfortunately the scalability of this powerful technique has not been demonstrated.

Automatic Inference. Our ownership inference is fully automatic and quite powerful. Automatic interprocedural annotation inferencing is also used in other software understanding tools in-

cluding Lackwit[23], Ajax[22] and AliasJava[1]. While all these tools support polymorphism, or context sensitivity, our algorithm is also flow-sensitive unlike the first two of these. Moreover, we do not know of any other inference algorithms that allow choice in ownership transfer in assignment statements and method invocations. This idea is also useful for solving other resource management problems.

Experimental Results. This paper includes experimental results demonstrating the success of our system in finding errors in large C and C++ programs. Very few experimental results have been reported in previous work involving ownership-based models. We are able to apply our system to large programs because the system assumes a flexible model, requires no user intervention, and has an efficient flow-sensitive and context-sensitive algorithm.

The GNU C++ compiler has a `-WeffC++` flag which warns of violations of Meyers's effective C++ rules, which include some rules to minimize memory leaks. The tool uses mainly syntactic analysis and tends to generate many false warnings. Fully automatic tools like PREfix[6] and Metal[12, 19] have been found to be effective in finding program errors. Although unsound and lacking a notion of object invariants, these tools have been shown to find many errors in large systems. Handling false warnings is a major issue in both of these systems. Experience with the PREfix system shows that it is important to prioritize the warnings so programmers can focus on the most likely errors. Metal statistically identifies common coding practice in a program and classifies deviations as errors.

We also prioritize warnings based on their likelihood of representing real errors. Moreover, because errors in our system appear as consistency violations, it is particularly important to attribute the errors to their source correctly. We do so by satisfying the more precise constraints first. Our experience with Clouseau confirms that a good constraint ordering is important to generating useful error reports.

7. CONCLUSIONS

This paper formalizes a practical object ownership model for managing memory. In this model, every object is pointed to by one and only one *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. In addition, a pointer-typed class member field either always or never owns its pointee at public method boundaries.

This model can be used to analyze existing code "as is" because it supports the normal semantics of assignments and method invocations. Namely, assignments may transfer ownership from the source to the destination variable, but are not required to do so. We capture the choice of ownership transfer with 0-1 integer linear inequalities. In addition, method interfaces also use 0-1 integer linear inequalities to capture the ownership relationships among parameters in a method. This representation supports a powerful form of polymorphism.

We have developed a flow-sensitive and context-sensitive algorithm that automatically infers field and method ownership signatures in C and C++ programs and identifies statements violating the model as errors. We have optimized the algorithm by using a sparse graph representation and a custom constraint solver to take advantage of the highly-structured 0-1 ownership constraints.

Our memory detection tool, Clouseau, is designed to help users isolate errors in a program. It tries to satisfy the more precise constraints first to prevent errors associated with the less precise constraints from propagating. Warnings are ranked so as to focus users on warnings most likely to lead to errors. Finally, the member field

ownerships and method signatures that the tool generates give the context of reported constraint violations and minimize the code inspection required to identify bugs from the warnings.

Our experimental results suggest that our algorithm is effective and efficient. It finds errors in each of the large software systems we experimented with, which include web servers, secure shell clients and servers, a chat client, object code tools and the SUIF compiler. The algorithm is practical; it analyzed over 50K lines of C++ code in about 9 minutes on a 2 GHz PC.

8. REFERENCES

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of OOPSLA 2002: Object-Oriented Programming Systems, Languages and Applications*, pages 311–330, November 2002.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of OOPSLA 2002: Object-Oriented Programming Systems, Languages and Applications*, pages 211–230, November 2002.

[3] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, pages 213–223, January 2003.

[4] J. Boyland. Alias burying: Unique variables without destructive reads. *Software–Practice and Experience*, 31(6):533–553, May 2001.

[5] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software–Practice and Experience*, 28(8):859–881, July 1998.

[6] 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.

[7] D. Clarke. An object calculus with ownership and containment. In *The Eighth International Workshop on Foundations of Object-Oriented Languages*, January 2001.

[8] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of OOPSLA 2002: Object-Oriented Programming Systems, Languages and Applications*, pages 292–310, November 2002.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA '98: Object-Oriented Programming Systems, Languages and Applications*, pages 48–64, October 1998.

[10] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Programming Languages*, pages 262–272, January 1999.

[11] N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Proceedings of the Static Analysis Symposium*, pages 115–134, July 2000.

[12] 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 *Proceedings of Eighteenth ACM Symposium on Operating System Principles*, pages 57–72, October 2001.

[13] D. Evans. Static detection of dynamic memory errors. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, May 1996.

[14] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.

[15] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 313–323, May 1998.

[16] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 70–80, May 2001.

[17] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, August 1986.

[18] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 282–293, June 2002.

[19] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, June 2002.

[20] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, December 1992.

[21] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *The 12th European Conference on Object-Oriented Programming*, pages 158–185, July 1998.

[22] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, November 2000.

[23] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, May 1997.

[24] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the Ninth European Symposium on Programming*, pages 366–381, April 2000.

[25] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[26] P. Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, April 1990.

[27] D. Walker and G. Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177–206, 2001.

[28] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

APPENDIX

A. STATIC TYPING RULES

This appendix includes all the static typing rules in our ownership type system. Figure 7 gives the typing rules for all statements in a method, other than those involving method invocations. Figure 8 gives the typing rules for method definitions and invocations.

Figure 9 gives the syntax and the rules of the static judgments. A program P is well-typed, $P \vdash \diamond$, if it has a well-typed constructor, destructor, methods and fields for each class in the program. A set of constraints C is good, $\vdash C$, if it is consistent. We define the restriction $C{\downarrow}\vec{\alpha}$ to be the resulting constraint after projecting away all variables but those in $\vec{\alpha}$.

(STMT-NULL)
$$\frac{D \vdash z \colon \rho_z \qquad \rho_z{}' \; fresh \qquad C' = C \wedge \{\rho_z{=}0\}}{B; D; C \vdash z = \texttt{NULL} \Rightarrow D[z{\mapsto}\rho_z{}']; C'}$$

(STMT-ASGN)
$$\frac{D \vdash y \colon \rho_y, z \colon \rho_z \qquad\qquad\qquad}{\rho_z{}', \rho_y{}' \; fresh \qquad C' = C \wedge \{\rho_z{=}0\} \wedge \{\rho_y{=}\rho_y{}' + \rho_z{}'\}}{B; D; C \vdash z = y \Rightarrow D[z{\mapsto}\rho_z{}', y{\mapsto}\rho_y{}']; C'}$$

(STMT-FLDASGN)
$$\frac{B \vdash \texttt{this} \colon c \qquad D \vdash y \colon \rho_y, \texttt{this}.f \colon \rho_f \qquad \rho_f{}', \rho_y{}' \; fresh}{C' = C \wedge \{\rho_f = 0\} \wedge \{\rho_y{=}\rho_y{}' + \rho_f{}'\} \qquad f \in \mathrm{F}(c)}{B; D; C \vdash \texttt{this}.f = y \Rightarrow D[\texttt{this}.f{\mapsto}\rho_f{}', y{\mapsto}\rho_y{}']; C'}$$

(STMT-FLDUSE)
$$\frac{B \vdash \texttt{this} \colon c \qquad D \vdash \texttt{this}.f \colon \rho_f, z \colon \rho_z \qquad \rho_z{}', \rho_f{}' \; fresh}{C' = C \wedge \{\rho_z = 0\} \wedge \{\rho_f{=}\rho_f{}' + \rho_z{}'\} \qquad f \in \mathrm{F}(c)}{B; D; C \vdash z = \texttt{this}.f \Rightarrow D[\texttt{z}{\mapsto}\rho_z{}', \texttt{this}.f{\mapsto}\rho_f{}']; C'}$$

(STMT-COMP)
$$\frac{B; D; C \vdash s_1 \Rightarrow D_1; C_1 \qquad B; D_1; C_1 \vdash s_2 \Rightarrow D_2; C_2}{B; D; C \vdash s_1 \cdot s_2 \Rightarrow D_2; C_2}$$

(STMT-IF)
$$\frac{B; D; C \vdash s_1 \Rightarrow D_1; C_1 \qquad B; D; C \vdash s_2 \Rightarrow D_2; C_2}{C' = C_1 \wedge C_2 \wedge \{D_1(x) = D_2(x) | x \in \mathrm{Dom}(D)\}}{B; D; C \vdash \texttt{if} \sim \texttt{then } s_1 \texttt{ else } s_2 \Rightarrow D_1; C'}$$

(STMT-WHILE)
$$\frac{B; D; C \vdash s \Rightarrow D_1; C_1}{C' = C \wedge C_1 \wedge \{D(x) = D_1(x) | x \in \mathrm{Dom}(D)\}}{B; D; C \vdash \texttt{while} \sim \texttt{do } s \Rightarrow D; C'}$$

Figure 7: Typing rules for statements

[METHOD-GOOD]

$$\text{MOT}_c(m) = [\alpha_\varsigma\ \vec{\alpha_x}\ \vec{\alpha_f}\ \vec{\alpha_f}'\ \alpha_r';\ C_m]$$

$\vec{f}{=}\text{F}(c)$ $\quad \rho_r, \vec{\rho_z}\ \textit{fresh}$ $\quad \text{this}: c, \vec{x}: \vec{t_x}, r: t_r, \vec{z}: \vec{t_z}; \text{this}: \alpha_\varsigma, \vec{x}: \vec{\alpha_x}, \text{this}.\vec{f}: \vec{\alpha_f}, r: \rho_r, \vec{z}: \vec{\rho_z}; \text{true} \vdash s \Rightarrow D; C$

$$D \vdash \text{this}: \rho_\varsigma', \vec{x}: \vec{\rho_x}', \text{this}.\vec{f}: \vec{\rho_f}', r: \rho_r', \vec{z}: \vec{\rho_z}'$$

$$\frac{C' = \{\rho_r{=}0\} \wedge \{\vec{\rho_z}{=}0\} \wedge C \wedge \{\rho_\varsigma'{=}0\} \wedge \{\vec{\rho_x}'{=}0\} \wedge \{\vec{\rho_f}'{=}\vec{\alpha_f}'\} \wedge \{\rho_r'{=}\alpha_r'\} \wedge \{\vec{\rho_z}'{=}0\} \quad \vdash C_m \quad C_m \leq C' \downarrow \alpha_\varsigma\ \vec{\alpha_x}\ \vec{\alpha_f}\ \vec{\alpha_f}'\ \alpha_r'}{\vdash m(\text{this}: c, \vec{x}: \vec{t_x}): t_r\ \{\text{scope}\ r: t_r, \vec{z}: \vec{t_z}\ \{s \cdot \text{return}\ r\}\}}$$

[CONSTR-GOOD]

$$\text{MOT}_c(\text{new}) = [\vec{\alpha_x}\ \vec{\alpha_f}'\ \alpha_r';\ C_m] \qquad \vec{f}{=}\text{F}(c) \qquad \rho_\varsigma, \vec{\rho_f}, \vec{\rho_z}\ \textit{fresh}$$

$\text{this}: c, \vec{x}: \vec{t_x}, \vec{z}: \vec{t_z}; \text{this}: \rho_\varsigma, \vec{x}: \vec{\alpha_x}, \text{this}.\vec{f}: \vec{\rho_f}, \vec{z}: \vec{\rho_z}; \text{true} \vdash s \Rightarrow D; C \qquad D \vdash \text{this}: \rho_\varsigma', \vec{x}: \vec{\rho_x}', \text{this}.\vec{f}: \vec{\rho_f}', \vec{z}: \vec{\rho_z}'$

$$\frac{C' = \{\rho_\varsigma{=}1\} \wedge \{\vec{\rho_f}{=}0\} \wedge \{\vec{\rho_z}{=}0\} \wedge C \wedge \{\rho_\varsigma'{=}\alpha_r'{=}1\} \wedge \{\vec{\rho_x}'{=}0\} \wedge \{\vec{\rho_f}'{=}\vec{\alpha_f}'\} \wedge \{\vec{\rho_z}'{=}0\} \quad \vdash C_m \quad C_m \leq C' \downarrow \vec{\alpha_x}\ \vec{\alpha_f}'\ \alpha_r'}{\vdash \text{new}_c(\vec{x}: \vec{t_x}): c\ \{\text{scope}\ \text{this}: c, \vec{z}: \vec{t_z}\ \{s \cdot \text{return}\ \text{this}\}\}}$$

[DESTR-GOOD]

$\text{MOT}_c(\text{delete}) = [\alpha_\varsigma\ \vec{\alpha_f};\ C_m] \qquad \vec{f}{=}\text{F}(c) \qquad \vec{\rho_z}\ \textit{fresh} \qquad \text{this}: c, \vec{z}: \vec{t_z}; \text{this}: \alpha_\varsigma, \text{this}.\vec{f}: \vec{\alpha_f}, \vec{z}: \vec{\rho_z}; \text{true} \vdash s \Rightarrow D; C$

$$\frac{D \vdash \text{this}: \rho_\varsigma', \text{this}.\vec{f}: \vec{\rho_f}', \vec{z}: \vec{\rho_z}' \qquad C' = \{\alpha_\varsigma{=}1\} \wedge \{\vec{\rho_z}{=}0\} \wedge C \wedge \{\rho_\varsigma'{=}1\} \wedge \{\vec{\rho_f}'{=}0\} \wedge \{\vec{\rho_z}'{=}0\} \qquad \vdash C_m \qquad C_m \leq C' \downarrow \alpha_\varsigma\ \vec{\alpha_f}}{\vdash \text{delete}_c(\text{this}: c)\ \{\text{scope}\ \vec{z}: \vec{t_z}\ \{s\}\}}$$

[STMT-INTCALL]

$B \vdash \text{this}: c \qquad \vec{f}{=}\text{F}(c) \qquad D \vdash \text{this}: \rho_\varsigma, \vec{x}: \vec{\rho_x}, \text{this}.\vec{f}: \vec{\rho_f}, r: \rho_r$

$\rho_\varsigma', \vec{\rho_x}', \vec{\rho_f}', \rho_r'\ \textit{fresh} \qquad \text{MOT}_c(m) = [\alpha_\varsigma\ \vec{\alpha_x}\ \vec{\alpha_f}\ \vec{\alpha_f}'\ \alpha_r';\ C_m] \qquad C_s = C_m[\sigma_\varsigma\ \vec{\sigma_x}\ \vec{\sigma_f}\ \vec{\sigma_f}'\ \sigma_r'\ \textit{fresh}/\alpha_\varsigma\ \vec{\alpha_x}\ \vec{\alpha_f}\ \vec{\alpha_f}'\ \alpha_r']$

$$\frac{C' = C \wedge \{\rho_r{=}0\} \wedge C_s \wedge \{\rho_\varsigma{=}\rho_\varsigma' + \sigma_\varsigma\} \wedge \{\vec{\rho_x}{=}\vec{\sigma_x}\} \wedge \{\vec{\rho_x}'{=}0\} \wedge \{\vec{\rho_f}{=}\vec{\sigma_f}\} \wedge \{\vec{\rho_f}'{=}\vec{\rho_f}'\} \wedge \{\sigma_r'{=}\rho_r'\}}{B; D; C \vdash r = \text{this}.m(\vec{x}) \Rightarrow D[\text{this} \mapsto \rho_\varsigma', \vec{x} \mapsto \vec{\rho_x}', \text{this}.\vec{f} \mapsto \vec{\rho_f}', r \mapsto \rho_r']; C'}$$

[STMT-EXTCALL]

$B \vdash y: c, \text{this}: c_1 \qquad \vec{f}{=}\text{F}(c) \qquad \vec{f_1}{=}\text{F}(c_1) \qquad D \vdash y: \rho_\varsigma, \vec{x}: \vec{\rho_x}, \text{this}.\vec{f_1}: \vec{\rho_{f_1}}, r: \rho_r$

$\rho_\varsigma', \vec{\rho_x}', \rho_r'\ \textit{fresh} \qquad \text{MOT}_c(m) = [\alpha_\varsigma\ \vec{\alpha_x}\ \vec{\alpha_f}\ \vec{\alpha_f}'\ \alpha_r';\ C_m] \qquad C_s = C_m[\sigma_\varsigma\ \vec{\sigma_x}\ \vec{\sigma_f}\ \vec{\sigma_f}'\ \sigma_r'\ \textit{fresh}/\alpha_\varsigma\ \vec{\alpha_x}\ \vec{\alpha_f}\ \vec{\alpha_f}'\ \alpha_r']$

$$\frac{C' = C \wedge \{\rho_r{=}0\} \wedge \{\vec{\rho_{f_1}}{=}\text{FOT}_{c_1}(\vec{f_1})\} \wedge C_s \wedge \{\rho_\varsigma{=}\rho_\varsigma' + \sigma_\varsigma\} \wedge \{\vec{\rho_x}{=}\vec{\sigma_x}\} \wedge \{\vec{\rho_x}'{=}0\} \wedge \{\vec{\sigma_f}{=}\vec{\sigma_f}'{=}\text{FOT}_c(\vec{f})\} \wedge \{\sigma_r'{=}\rho_r'\}}{B; D; C \vdash r = y.m(\vec{x}) \Rightarrow D[y \mapsto \rho_\varsigma', \vec{x} \mapsto \vec{\rho_x}', r \mapsto \rho_r']; C'}$$

[STMT-CONSTR]

$$B \vdash \text{this}: c_1 \qquad \vec{f}{=}\text{F}(c) \qquad \vec{f_1}{=}\text{F}(c_1)$$

$D \vdash \vec{x}: \vec{\rho_x}, \text{this}.\vec{f_1}: \vec{\rho_{f_1}}, r: \rho_r \qquad \vec{\rho_x}', \rho_r'\ \textit{fresh} \qquad \text{MOT}_c(\text{new}) = [\vec{\alpha_x}\ \vec{\alpha_f}'\ \alpha_r';\ C_m] \qquad C_s = C_m[\vec{\sigma_x}\ \vec{\sigma_f}'\ \sigma_r'\ \textit{fresh}/\vec{\alpha_x}\ \vec{\alpha_f}'\ \alpha_r']$

$$\frac{C' = C \wedge \{\rho_r{=}0\} \wedge \{\vec{\rho_{f_1}}{=}\text{FOT}_{c_1}(\vec{f_1})\} \wedge C_s \wedge \{\vec{\rho_x}{=}\vec{\sigma_x}\} \wedge \{\vec{\rho_x}'{=}0\} \wedge \{\vec{\sigma_f}'{=}\text{FOT}_c(\vec{f})\} \wedge \{\sigma_r'{=}\rho_r'\}}{B; D; C \vdash r = \text{new}_c(\vec{x}) \Rightarrow D[\vec{x} \mapsto \vec{\rho_x}', r \mapsto \rho_r']; C'}$$

[STMT-DESTR]

$B \vdash y: c, \text{this}: c_1 \qquad \vec{f}{=}\text{F}(c) \qquad \vec{f_1}{=}\text{F}(c_1) \qquad D \vdash y: \rho_\varsigma, \text{this}.\vec{f_1}: \vec{\rho_{f_1}}, \qquad \rho_\varsigma'\ \textit{fresh} \qquad \text{MOT}_c(\text{delete}) = [\alpha_\varsigma\ \vec{\alpha_f};\ C_m]$

$$\frac{C_s = C_m[\sigma_\varsigma\ \vec{\sigma_f}\ \textit{fresh}/\alpha_\varsigma\ \vec{\alpha_f}] \qquad C' = C \wedge \{\vec{\rho_{f_1}}{=}\text{FOT}_{c_1}(\vec{f_1})\} \wedge C_s \wedge \{\rho_\varsigma{=}\rho_\varsigma' + \sigma_\varsigma\} \wedge \{\vec{\sigma_f}{=}\text{FOT}_c(\vec{f})\}}{B; D; C \vdash y.\text{delete}_c() \Rightarrow D[y \mapsto \rho_\varsigma']; C'}$$

Figure 8: Good method definition and invocation typing rules

[PROGRAM-GOOD]

$$\forall c \in \text{CL}(P), \forall f \in \text{F}(c), m \in \text{M}(c)$$

$$\frac{\text{FOT}_c(f) = [\rho; \rho = \kappa]\ \kappa \in \{0,1\} \qquad \vdash m(\text{this}: c, \vec{x}: \vec{t_x}): t_r\ \{...\} \qquad \vdash \text{new}_c(\vec{x}: \vec{t_x}): c\ \{...\} \qquad \vdash \text{delete}_c(\text{this}: c)\ \{...\}}{P \vdash \diamond}$$

[CONSTRAINT-GOOD]

$$\frac{C \nvdash \text{false}}{\vdash C}$$

Figure 9: Well-typed programs