Reference Count Analysis With Shallow Aliasing

Akash Lal[1] akash@cs.wisc.edu
G. Ramalingam grama@microsoft.com

May 2009

Technical Report
MSR-TR-2009-61

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

[1]Work done when the author was an intern at Microsoft Research, India.

1 Introduction

Resource management is a key problem in system software, especially in the presence of concurrency. Reference counting is a commonly used technique for resource management. Typically, the reference count field of an object keeps track of the number of references of a certain kind to that object. For example, for garbage collection, the field keeps track of the number of pointers pointing to the given object. Reference counting is also used to implement reader-writer locks in the presence of concurrency: read locks allow a number of threads to concurrently read an object, while a thread must acquire an exclusive lock to update the object. In this context, a reference count field is used to keep track of the number of readers holding a read-lock on the resource.

A reference count field is of type `integer` and is manipulated using increment and decrement operations. For example, when a new client acquires a resource, the concerned reference count field is incremented by one. When the client gives up the resource, the field is decremented by one. It is safe for a client to increment the same reference count multiple times, as long as the client performs a matching number of the decrements of the reference count at the appropriate time. (When used to implement a locking mechanism, this feature allows a thread to acquire a lock on a resource without having to check if it already holds a lock on that resource, which is often convenient.)

In the garbage-collection context, an object whose reference count is zero is garbage and may be reclaimed. In the reader-writer lock scenario, an object whose reference count is zero does not have any readers, and a writer may acquire a write-lock on the corresponding object.

One key correctness criterion in the use of reference counts is that the increment and decrement operations must be well-matched. If some computation increments a reference count but omits to perform a matching decrement, the reference count may never reach zero. This can not only lead to a degradation of the performance of an application, but it can also have more human-visible effects. For example, if reference counts are used to track the number of applications currently accessing a file or a folder, a user may be prevented from deleting a file or folder whose reference count is positive. Thus, if the reference count is not accurate in this context, it can lead to undesirable behavior.

Conversely, an extra decrement operation may lead to *dangling references* as objects may be freed prematurely while they are still being used or a writer may acquire a write-lock on an object that is still being used by readers.

In this paper we are interested in statically verifying that a given program uses reference counts correctly. Reference count analysis is typically amenable to modular analysis of threads using assume-guarantee reasoning: one can verify that a concurrent program follows the reference count protocol and enjoys certain safety properties by verifying that every thread follows the reference count protocol correctly. Hence, we will restrict our attention to sequential programs in this paper.

The property that we are interested in verifying is that a program (a thread) performs an equal number of increment and decrement operations on every object. We call this the *reference count property*.

In this paper, we give a polynomial time algorithm for verifying this property when the program is only allowed to have *shallow pointers*, i.e., pointers are not allowed to point to other pointers. We also show that in the presence of general (non-shallow) pointers, but in the absence of procedures and recursive data structures, the problem is PSPACE-complete.

The reference count property is similar to a *typestate* property. In typestate verification, one wishes to verify that the sequence of operations performed on any given object belongs to a given regular language. The reference count verification is similar, except that the set of sequences of operations allowed on an object is not a regular set, but a context-free set. Thus, we cannot use a finite-state machine to capture the states of an object for purposes of verification (unlike in typestate verification).

This seems to suggest that reference count verification would be harder than typestate verification. However, we show that in the presence of shallow pointers, reference count verification is in P, whereas certain classes of typestate verification problems are known to be PSPACE-complete [3].

We solve the problem of reference count verification by giving a reduction from a program P with shallow pointers to an *affine program* A_P. An affine program only has integer-valued variables (no pointers) and the right-hand side of each assignment is a linear polynomial in the program variables. Such programs can be analyzed in polynomial time to infer all linear equalities that hold between the program variables [5, 7]. This analysis is called affine-relation analysis. We show how backward affine-relation analysis on A_P can be used for reference count verification of P. We are not aware of any other reduction from a program with pointers to one without them.

2 A Polynomial Time Intraprocedural Algorithm

We begin by describing our algorithm for programs with a single procedure. We extend the approach to handle multiple procedures (and procedure calls) in §3.

The Problem

The Language. Consider programs with a single `main` procedure, and a fixed number of global variables. The programs are assumed to manipulate objects of a single type `T`, which has just one field `refcount` of type `integer`. The variables are assumed to be shallow pointers of type `T*`. Programs can have the usual control flow. We make the common assumption that all paths are executable, i.e., the branch conditions are non-deterministic. A program can have five kinds of statements:

```
x1 = new;   x2 = new;   x3 = new;   x4 = new;
if(...){
    x3 = x1;   x4 = x2;
} else {
    x3 = x2;   x4 = x1;
}
if(...){
    x1 → refcount ++;   x2 → refcount ++;
    x3 → refcount --;   x4 → refcount --;
}
```

Figure 1: An example program with four global variables $\mathtt{x}_1, \cdots, \mathtt{x}_4$.

Statement	**Meaning**
`x = y`	Copy from `y` to `x`
`x → refcount ++`	Increment the `refcount` field of object pointed to by `x`
`x → refcount --`	Decrement the `refcount` field of object pointed to by `x`
`x = new`	`x` points to a newly allocated object
`x = null`	`x` is made `null`

An example program is shown in Fig. 1. The program has four execution paths, and the reader can verify that along all those paths the reference count of an object is zero when it goes out of scope.

Concrete Semantics. Fix a program P. Let `Var` be the set of global variables of P. The state of program P during execution consists of the program counter as well as a triple $\langle \mathcal{H}, \upsilon, \kappa \rangle$, where $\mathcal{H}$ represents the set of heap-allocated objects, $\upsilon : \mathtt{Var} \mapsto \mathcal{H}$ is a map from variables to the objects they point to, and $\kappa : \mathcal{H} \mapsto \mathit{int}$ is a map from objects to the value of their `refcount` field. We will use the shorthand notation $rc(\mathtt{x}, \sigma)$ to represent the `refcount` value of the object that `x` points to in state σ. Thus, $rc(\mathtt{x}, \langle \mathcal{H}, \upsilon, \kappa \rangle)$ is equal to $\kappa(\upsilon(\mathtt{x}))$.

We simplify the semantics by treating the `null` value as a constant representing the address of a special object o_{null}. We will assume that all variables are initialized to `null` when program execution begins. Thus, the initial program state is represented by the triple $\langle \{o_{\text{null}}\}, \lambda\mathtt{x}.o_{\text{null}}, \lambda y.0 \rangle$. All newly allocated objects have their `refcount` field initialized to zero. We omit a formal definition of the semantics of the various statements as they are quite standard.

In this paper we will assume that the given program does not contain any null pointer dereferences or dereferences of uninitialized variables. Note that the above simplification has the effect of converting dereferences of uninitialized variables into dereferences of a null pointer. In the absence of multi-level pointers (pointers to pointers or pointers to objects containing pointers), potential null pointer dereferences can be easily identified.

The Problem. The property we wish to check for is that the `refcount` value of every object is zero in any final state produced by the execution of the program.

A Finite But Precise Abstraction.

We first describe a finite abstraction that can be used to precisely check for the above property.

Because the name or identity of an object is immaterial, we can use a (storeless) abstraction of the program state of the form $\{(S_1, v_1), (S_2, v_2), \cdots, (S_n, v_n)\}$, where $\{S_i\}_{i=1}^n$ forms a partition of `Var` and each v_i is an integer. This state represents the fact that all variables in S_i point to the same object and the `refcount` of that variable has value v_i. For example, the state after execution of line 3 in Fig. 1 is $\{(\{\mathtt{x}_1, \mathtt{x}_3\}, 0), (\{\mathtt{x}_2, \mathtt{x}_4\}, 0)\}$.

Note that the above abstraction also omits information about *unreachable* objects in the heap: objects that are not pointed to by any program variable. We say an object *becomes unreachable* due to the execution of a program statement if the object is pointed to by some variable in the state before the statement execution but not in the state after the statement execution. Obviously, the `refcount` of an object can never change once it becomes unreachable.

Thus, we can perform the desired verification with the storeless abstraction by (i) Checking that the `refcount` of any object is zero when it becomes unreachable, and (ii) Checking that the `refcount` of all reachable objects in any final state is zero.

The storeless abstraction is, however, not a finite abstraction. A program's execution can still produce an unbounded number of states with this abstraction. However, we can still do the reference count verification precisely as follows. We say that two states $\{(S_1, v_1), (S_2, v_2), \cdots, (S_n, v_n)\}$ and $\{(R_1, w_1), (R_2, w_2), \cdots, (R_n, w_n)\}$ have the *same aliasing configuration* if the partitions $\{S_i\}_{i=1}^n$ and $\{R_i\}_{i=1}^n$ are identical. Note that if two states arise at the same program point with same aliasing configuration but different values of `refcount`, then the program is bound to have an error. Suppose the two states are $\{(S_1, v_1), \cdots\}$ and $\{(S_1, v_1'), \cdots\}$ with $v_1 \neq v_1'$, and the program point is p. Let the concerned objects in the two states, which are pointed to by all variables in S_1, be o and o', respectively. For the program to be correct, there must be a path starting at p that decrements the `refcount` of o by v_1 when it goes out of scope. Because the program is never allowed to look at the value of `refcount`, the very same path will decrement the value of o_2 by v_1, leaving its `refcount` at $v_1' - v_1 \neq 0$ when it goes out of scope. Hence, the reference count property is not satisfied. This shows that an analysis only needs to track states with different aliasing configurations at any given program point. Because the number of such states is bounded, the problem is decidable.

The above bound is still exponential in the number of variables. To obtain a polynomial time algorithm, we need further ways of cutting down on the information to be tracked.

Analyzing A Single Path

We first present an algorithm for verifying that a single execution path ρ satisfies the reference count property. The approach presented here may seem unnecessarily complex as a single path can be analyzed quite easily. However, as we will show later, this approach has the advantage that it can be generalized in a straightforward fashion to analyze a whole program.

Let $\sigma_1 = \langle \mathcal{H}_1, \upsilon_1, \kappa_1 \rangle$ denote any state. Let $\sigma_2 = \langle \mathcal{H}_2, \upsilon_2, \kappa_2 \rangle$ denote the state produced by the execution of path ρ starting from state σ_1. Let o be any object in the initial state σ_1. We define $\Delta(\rho, \sigma_1, o)$ to be $\kappa_2(o) - \kappa_1(o)$. Thus, $\Delta(\rho, \sigma_1, o)$ captures the effect of executing ρ, starting at state σ_1, on the reference count of o.

Assume that the given program has n global variables $\mathtt{x}_1$ to $\mathtt{x}_n$. Let σ_u be the state where each variable $\mathtt{x}_i$ points to a distinct object o_i with a reference count of 0: thus, it corresponds to the state $\{(\mathtt{x}_1, 0), (\{\mathtt{x}_2\}, 0), \cdots, (\{\mathtt{x}_n\}, 0)\}$. We define $\Delta(\rho, \mathtt{x}_i)$ to be $\Delta(\rho, \sigma_u, o_i)$.

The reason that $\Delta(\rho, \mathtt{x}_i)$ is of interest is that it can be used to compute any $\Delta(\rho, \sigma, o)$ as shown below. Let $\mathit{refs}(\sigma, o)$ denote the set of all variables $\mathtt{x}_j$ that point to o in state σ.

Lemma 1. $\Delta(\rho, \sigma, o) = \sum_{x_j \in \mathit{refs}(\sigma, o)} \Delta(\rho, x_j)$.

Proof. We proceed by induction on the length of ρ, denoted as $|\rho|$.

Base case. When $|\rho| = 0$, i.e., the path is empty, both sides of the above equation evaluate to zero and the lemma holds trivially.

Inductive case. We do a case analysis on the first statement `st` of ρ. Let the suffix of ρ after the first statement be ρ'. Let σ' denote the program state after execution of `st`. Let c_i and c'_i denote $\Delta(\rho, \mathtt{x}_i)$ and $\Delta(\rho', \mathtt{x}_i)$ respectively.

Case 1. `st` is $\mathtt{x}_1 \rightarrow$ `refcount++`. In this case, $c_1 = c'_1 + 1$ and $c_i = c'_i$ for $2 \leq i \leq n$. Furthermore, $\Delta(\rho, \sigma, o)$ is equal to $\Delta(\rho', \sigma', o) + 1$ if $\mathtt{x}_1 \in \mathit{refs}(\sigma, o)$ and $\Delta(\rho, \sigma, o)$ is equal to $\Delta(\rho', \sigma', o)$ if $\mathtt{x}_1 \notin \mathit{refs}(\sigma, o)$ In either case, the lemma holds. This establishes the inductive case. When `st` is $\mathtt{x}_1 \rightarrow$ `refcount--`, we can use a similar argument.

The execution of other types of statements does not change the reference count of any object. As a result, $\Delta(\rho, \sigma, o)$ is equal to $\Delta(\rho', \sigma', o)$, which is itself equal to $\sum_{\mathtt{x}_j \in \mathit{refs}(\sigma', o)} \Delta(\rho', \mathtt{x}_j)$ by the inductive hypothesis. The result will follow if we can show that $\sum_{\mathtt{x}_j \in \mathit{refs}(\sigma', o)} \Delta(\rho', \mathtt{x}_j) = \sum_{\mathtt{x}_j \in \mathit{refs}(\sigma, o)} \Delta(\rho, \mathtt{x}_j)$, which we do below.

Case 2. `st` is $\mathtt{x}_1 =$ `new`. In this case, note that $\mathit{refs}(\sigma', o) = \mathit{refs}(\sigma, o) - \{\mathtt{x}_1\}$ for any object o in σ. Furthermore, $\Delta(\rho, \mathtt{x}_1) = 0$, and $\Delta(\sigma', \mathtt{x}_i) = \Delta(\sigma, \mathtt{x}_i)$ for any $\mathtt{x}_i \in \mathit{refs}(\sigma, o) - \{\mathtt{x}_1\}$. Hence, $\sum_{\mathtt{x}_j \in \mathit{refs}(\sigma', o)} \Delta(\rho', \mathtt{x}_j) = \sum_{\mathtt{x}_j \in \mathit{refs}(\sigma, o)} \Delta(\rho, \mathtt{x}_j)$.

Case 3. $\texttt{st}$ is $\texttt{x}_1 = \texttt{x}_2$. First, let us determine the values of c_i by considering the execution of this statement on σ_u, which produces the state $\sigma'_u = \{(\{\texttt{x}_1, \texttt{x}_2\}, 0), (\{\texttt{x}_3\}, 0), \cdots, (\{\texttt{x}_n\}, 0)\}$. Using the inductive hypothesis, the effect of executing ρ' on o_1 is 0 (because it is not in scope), on o_2 is $c'_1 + c'_2$, on o_i, $i > 2$ is c'_i. Therefore, $c_1 = 0, c_2 = c'_1 + c'_2, c_i = c'_i$, $i > 2$. We need to show that $\sum_{\texttt{x}_j \in \mathit{refs}(\sigma', o)} \Delta(\rho', \texttt{x}_j) = \sum_{\texttt{x}_j \in \mathit{refs}(\sigma, o)} \Delta(\rho, \texttt{x}_j)$, We consider the following four subcases. Case $\texttt{x}_1 \notin \mathit{refs}(\sigma, o)$ and $\texttt{x}_2 \notin \mathit{refs}(\sigma, o)$: the result follows trivially. Case $\texttt{x}_1 \notin \mathit{refs}(\sigma, o)$ and $\texttt{x}_2 \in \mathit{refs}(\sigma, o)$: in this case, we have $\mathit{refs}(\sigma', o) = (\mathit{refs}(\sigma, o) \cup \{\texttt{x}_1\}) \supseteq \{\texttt{x}_2\}$; hence, we have $c'_1 + c'_2$ on the left-hand side and c_2 on the right-hand side (apart from the common terms) and the result follows. Case $\texttt{x}_1 \in \mathit{refs}(\sigma, o)$ and $\texttt{x}_2 \notin \mathit{refs}(\sigma, o)$: in this case, we have $\texttt{x}_2 \notin \mathit{refs}(\sigma', o) = \mathit{refs}(\sigma, o) - \{\texttt{x}_1\}$ and the result follows. Case $\texttt{x}_1 \in \mathit{refs}(\sigma, o)$ and $\texttt{x}_2 \in \mathit{refs}(\sigma, o)$: in this case, we have $\mathit{refs}(\sigma', o) = \mathit{refs}(\sigma, o) \supseteq \{\texttt{x}_1, \texttt{x}_2\}$ and the result follows. $\square$

Lemma 1 says that we only need to consider the aliasing scenario σ_u in which all variables are unaliased. The effect of a program path on other states can be calculated from the effect of that path on σ_u.

Given a path ρ, it is relatively straightforward to compute the values of $\Delta(\rho, \texttt{x}_i)$ for every variable $\texttt{x}_i$ inductively (along the lines in the inductive proof of Lemma 1). For reasons that will become clear soon, we will actually construct a (straight-line) program that can compute these values as follows. For each variable $\texttt{x}_i$ in ρ, we introduce an integer-valued variable X_i in the constructed program that is used to compute $\Delta(\rho, \texttt{x}_i)$.

Given path ρ, let ρ_T be the path obtained by replacing every statement S in ρ by the corresponding code-fragment S_T shown in Fig. 3. We will make the simplifying assumption that the first vertex in ρ is a unique *entry* vertex and that the last vertex in ρ is a unique *exit* vertex, which are also transformed as indicated in Fig. 3. We define ρ_R to be the path (program) obtained by reversing ρ_T. Thus, if ρ is $\mathit{entry}; S_1; \cdots S_k; \mathit{exit}$, then ρ_R is $\mathit{exit}_T; S_{kT}; \cdots S_{1T}; \mathit{entry}_T$.

We now explain how to interpret the transformed statement S_T. (For now, ignore the assertions embedded in S_T.) Assume that the path ρ consists of statement S followed by path ρ'. The statement S_T shows how the values of $\Delta(\rho, \texttt{x}_i)$ (represented by X_i on the left-hand side of assignments) can be computed from the values of $\Delta(\rho', \texttt{x}_i)$ (represented by X_i on the right-hand side of the assignments). The program ρ_R is an imperative program that computes the same values for the whole path.

An example is shown in Fig. 2. Interesting facts to note from this example are: the value of X_1 is 0 at line 3 because $\texttt{x}_1$ starts pointing to a new object, and the old object is not touched by further dereferences to $\texttt{x}_1$; at line 1 we get to know the fact that $\texttt{x}_2$ and $\texttt{x}_1$ are aliased, and the value of X_1 captures the dereferences made through both $\texttt{x}_1$ and $\texttt{x}_2$. The value of X_2 is made zero because, again, $\texttt{x}_2$ loses reference to the original object it pointed to.

Lemma 2. *Let ρ be any program path. (a) The final value of variable X_i after the execution of ρ_R is $\Delta(\rho, \texttt{x}_i)$. (b) The execution of ρ satisfies the reference*

1.	$\mathtt{x}_2 = \mathtt{x}_1$	$X_1 = 2, X_2 = 0$
2.	$\mathtt{x}_1 \rightarrow$ `refcount ++`	$X_1 = 1, X_2 = 1$
3.	$\mathtt{x}_1 =$ `new`	$X_1 = 0, X_2 = 1$
4.	$\mathtt{x}_1 \rightarrow$ `refcount --`	$X_1 = -1, X_2 = 1$
5.	$\mathtt{x}_2 \rightarrow$ `refcount ++`	$X_1 = 0, X_2 = 1$
6.		$X_1 = 0, X_2 = 0$

Figure 2: A program path and the values of X_i. For each row, the third column denotes the values of X_i just before the program statement.

Statement S	**Statement** S_T
$\mathtt{x}_i = \mathtt{x}_j$	$X_j := X_i + X_j; X_i := 0$
$\mathtt{x}_i \rightarrow$ `refcount ++`	$X_i := X_i + 1$
$\mathtt{x}_i \rightarrow$ `refcount --`	$X_i := X_i - 1$
$\mathtt{x}_i =$ `new`	`assert` $X_i == 0$; $X_i := 0$
entry	`forall` i. `assert` $X_i == 0$
exit	`forall` i. $X_i := 0$

Figure 3: Updating the values of X_i according to program statements.

count property iff the execution of ρ_R satisfies all the assertions embedded in it.

Proof. The proof of (a) follows directly from Lemma 1.

Part (b) shows how we can exploit property (a) to check for the reference count property. To verify the reference count property, we must ensure that for any new object o allocated during the execution of ρ, the reference count value of o at the end of execution of ρ is zero. Consider any statement S : $\mathtt{x}_i =$ `new` in ρ that creates a new object o. Let ρ' represent the suffix of the program path ρ *after* this allocation statement. We can restate the requirement as: we need to verify that $\Delta(\rho', \mathtt{x}_i)$ is zero. It follows from (a) that the assertion "`assert` $X_i == 0$" inserted as part of the transformed statement S_T is equivalent to this check. □

For example, if the path shown in Fig. 2 ends at program exit, then the reference count property is not satisfied on the path: $X_1 = -1$ right after the execution of line 3, i.e., the `refcount` of that object is -1 when it goes out of scope.

Analyzing A Single-Procedure Program

We now show how the approach presented for analyzing a single path can be generalized to analyze whole programs.

Let P be a given program. We construct an affine program P_R along the same lines presented earlier. Let G be a control-flow graph representation of P with each vertex representing a single statement. Let G_T represent the control-flow graph obtained by replacing every statement s in G by the transformed

```
$X_1 := 0; X_2 := 0; X_3 := 0; X_4 := 0;$ // *initialization*
if(...){
    $X_4 := X_4 - 1; X_3 = X_3 - 1;$ // *line* 9
    $X_2 := X_2 + 1; X_1 = X_1 + 1;$ // *line* 8
}
if(...){
    $X_1 := X_1 + X_4; X_4 := 0; X_2 := X_2 + X_3; X_3 := 0;$ // *line* 5
} else {
    $X_2 := X_2 + X_4; X_4 := 0; X_1 := X_1 + X_3; X_3 := 0;$ // *line* 3
}
$\text{assert}(X_4 == 0); \text{assert}(X_3 == 0); \text{assert}(X_2 == 0); \text{assert}(X_1 == 0);$ // *line* 1
```

Figure 4: Affine program obtained for the one in Fig. 1. The comments show the lines in Fig. 1 from which the affine statements were obtained.

statement s_T as shown in Fig. 3. We define P_R to be the program obtained by reversing the control-flow edges of G_T.

Theorem 1. *A given program P satisfies the reference count property iff if the affine program P_R satisfies all its assertions.*

Proof. Follows directly from Lemma 2. □

For example, the affine program obtained for the program in Fig. 1 is shown in Fig. 4, along with the assertions that need to be verified.

Since P_R is an affine program, we can verify if P_R satisfies all its assertions (which are all affine assertions) using the previous results [5, 7] for reasoning about affine programs.

3 Polynomial Time Interprocedural Algorithm

In this section, we give a polynomial time algorithm for verification of the reference count property on programs, as defined in the previous section, but with multiple procedures, and procedure call statements. A procedure can have local variables, but no parameters (parameter passing can be implemented using global variables).

The algorithm is similar to the one described in the previous section. We convert a program P into an affine program A_P: each procedure of P results in a procedure of A_P with the reversed control flow. A_P has one global variable X_i for each global variable $\mathtt{x}_i$ of P, and one local variable X_j for each local variable $\mathtt{x}_j$ of P. The statements are transformed using Fig. 3, and procedure calls remain the same. The global variables of A_P are initialized with 0 at the beginning of the program, and the local variables are initialized to zero at the beginning of their procedure. Next, we verify assertions $X_i == 0$ in A_P at each program point that originally held $\mathtt{x}_i$ = `new`. An example is shown in Fig. 5.

```
main(){
    fnew();
    $x_5$ = new;
    if(...){
        fcopy1();
    } else {
        fcopy2();
    }
    if(...){
        fref();
    }
    $x_5 \rightarrow$ refcount++;
    $x_5 \rightarrow$ refcount--;
}
fnew(){
    $x_1$ = new; $x_2$ = new;
    $x_3$ = new; $x_4$ = new;
}
fcopy1(){
    $x_3 = x_1; x_4 = x_2$;
}
fcopy2(){
    $x_3 = x_2; x_4 = x_1$;
}
fref(){
    $x_1 \rightarrow$ refcount++;
    $x_2 \rightarrow$ refcount++;
    $x_3 \rightarrow$ refcount--;
    $x_4 \rightarrow$ refcount--;
}
```

```
main(){
    $X_1 := 0; X_2 := 0; X_3 := 0; X_4 := 0; X_5 := 0;$
    $X_5 := X_5 - 1;$
    $X_5 := X_5 + 1;$
    if(...){
        fref();
    }
    if(...){
        fcopy2();
    } else {
        fcopy1();
    }
    assert($X_5 == 0$);
    fnew();
}
fnew(){
    assert($X_4 == 0$); assert($X_3 == 0$);
    assert($X_2 == 0$); assert($X_1 == 0$);
}
fcopy1(){
    $X_1 = X_1 + X_3; X_3 := 0; X_2 = X_2 + X_4; X_4 = 0;$
}
fcopy2(){
    $X_2 = X_2 + X_3; X_3 := 0; X_1 = X_1 + X_4; X_4 = 0;$
}
fref(){
    $X_4 := X_4 - 1;$
    $X_3 := X_3 - 1;$
    $X_2 := X_2 + 1;$
    $X_1 := X_1 + 1;$
}
```

Figure 5: An example program, and the affine program obtained from it. Variables $\mathtt{x}_1, \cdots, \mathtt{x}_4, X_1, \cdots, X_4$ are global and variables $\mathtt{x}_5$ and X_5 are local to `main`.

All such assertions hold in A_P if and only if the reference count property holds in P.

The proof for the correctness of the above algorithm proceeds on the same lines as before, and Lemma 1 is extended to interprocedural paths.

We extend the program state to talk about the program stack as well. Let $\mathtt{Var}_G = \{\mathtt{g}_1, \cdots, \mathtt{g}_n\}$ be the set of global variables and $\mathtt{Var}_L = \{\mathtt{l}_1, \cdots, \mathtt{l}_m\}$ be the set of local variables (we assume that all procedures have the same number of local variables). A program state $\langle \mathcal{H}, \mathcal{S}, \upsilon_G, \upsilon_L, \kappa \rangle$ consists of a set $\mathcal{H}$ whose elements represent heap-allocated objects, a set $\mathcal{S} = \{1, 2, \cdots, 2\}$ whose elements represent activation records, a mapping υ_G from $\mathtt{Var}_G$ (the global variables) to $\mathcal{H}$, a mapping υ_L from $\mathcal{S} \times \mathtt{Var}_L$ (a stack of local variables, with one copy of the local variables for each activation record) to $\mathcal{H}$, and a map κ from objects in $\mathcal{H}$ to their `refcount` value.

We define the set of variable instances in a state $\sigma = \langle \mathcal{H}, \mathcal{S}, \upsilon_G, \upsilon_L, \kappa \rangle$ to be the set $\mathtt{Var}_G \cup \mathcal{S} \times \mathtt{Var}_L$. Let $\mathit{refs}(\sigma, o)$ denote the set of all variable instances that point to o in state σ.

The definition of Δ from the intraprocedural case can be generalized to the interprocedural case in a straightforward fashion. In this generalization, we will restrict ourselves to *valid terminating interprocedural paths*: these are interprocedural paths containing matching call and return edges, as well as some set of unmatched return edges. (These correspond to suffixes of valid interprocedural paths from program entry to program exit.)

Let $\sigma_1 = \langle \mathcal{H}_1, \mathcal{S}_1, \upsilon_{G1}, \upsilon_{L1}, \kappa_1 \rangle$ denote any interprocedural state and let ρ be any valid terminating interprocedural path such that the number of activation records in σ_1 is one more than the number of unmatched return edges in ρ. We say that σ_1 and ρ are *compatible* in this case. Let $\sigma_2 = \langle \mathcal{H}_2, \mathcal{S}_2, \upsilon_{G2}, \upsilon_{L2}, \kappa_2 \rangle$ denote the state produced by the execution of path ρ starting from state σ_1. Let o be any object in the initial state σ_1. We define $\Delta(\rho, \sigma_1, o)$ to be $\kappa_2(o) - \kappa_1(o)$. Thus, $\Delta(\rho, \sigma_1, o)$ captures the effect of executing ρ, starting at state σ_1, on the reference count of o.

Given a valid terminating interprocedural path ρ, let σ_u^ρ be the state that is compatible with ρ where each variable instance $\mathtt{x}_i$ points to a distinct object o_i with a reference count of 0. (The state σ_u^ρ is uniquely determined, upto isomorphism, if we consider only states without garbage, *i.e.*, objects that are not pointed-to by any variable instance.) We define $\Delta(\rho, \mathtt{x}_i)$ to be $\Delta(\rho, \sigma_u^\rho, o_i)$.

Lemma 3. $\Delta(\rho, \sigma, o) = \sum_{x_j \in \mathit{refs}(\sigma, o)} \Delta(\rho, x_j)$.

Proof. The proof proceeds by an induction on the length of ρ. The base case, when $|\rho| = 0$, is trivial. For the inductive case, we do a case analysis on the first statement of ρ. The cases when the first statement is a `refcount` increment or a decrement, or a copy $\mathtt{x}_i = \mathtt{x}_j$ or an allocation, are exactly same as for Lemma 1. We only need to consider the cases when the first statement is a procedure call or a procedure return. Let ρ' be the suffix of ρ after its first statement.

Case 1. Suppose that the first statement is a procedure call. We assume that all local variables are initialized to null at procedure entry. Because we have removed assignments to null, the local variables are assumed to be initialized to a special object `NullObj`. Thus, the variables in σ_u (which excludes the local variables of the new activation record) remain unaliased. In this case, $\Delta(\rho, \mathtt{x}) = \Delta(\rho', \mathtt{x})$ for all variables in σ_u.

When ρ is executed from $\{(S, o, v), \cdots\}$, the execution of the first statement does not affect object o, i.e., it produces another state of the same form (with more instantiations of local variables). By induction hypothesis, the execution of ρ' changes `refcount` of o to $v + \sum_{\mathtt{x} \in S} \Delta(\rho', \mathtt{x}) = v + \sum_{\mathtt{x} \in S} \Delta(\rho, \mathtt{x})$.

Case 2. Suppose that the first statement is a procedure return. Let L be the instantiation of the local variables that corresponds to the top of the stack (before the return is executed). Then we know that $\Delta(\rho, \mathtt{x}) = \Delta(\rho', \mathtt{x})$ for all variables $\mathtt{x}$ in σ_u but not in L. For $\mathtt{x} \in L$, $\Delta(\rho, \mathtt{x}) = 0$ because the objects pointed to by them immediately go out of scope.

Starting from state $\{(S, o, v), \cdots\}$ leads to state $\{(S - L, o, v), \cdots\}$ after the procedure return is executed. By induction hypothesis, the `refcount` of o when it goes out of scope is $v + \sum_{\mathtt{x} \in S-L} \Delta(\rho', \mathtt{x}) = v + \sum_{\mathtt{x} \in S} \Delta(\rho, \mathtt{x})$. $\square$

4 Intractable Problem Extensions

The problem of reference count verification in the presence of non-shallow pointers (but without recursive types or procedures) is PSPACE-hard. The reduction follows directly from the problem of must-alias analysis. From [8], we know that the problem of finding out that two variables are must aliases at a program point is PSPACE-hard. A must-alias question can be encoded using reference counts as follows.

Lemma 4. *Variables x_1 and x_2 are must aliases at a program point if and only if the the program satisfies the reference count property after the insertion of statements $x_1 \rightarrow$ `refcount++` and $x_2 \rightarrow$ `refcount--` at that program point.*

Theorem 2. *Verifying that the reference counts of all objects are zero at the end of execution is PSPACE-hard in the presence of two-level pointers.*

Note that the reference count problem is a particular form of affine analysis (in the presence of shallow pointers). However, other simple forms of affine analysis are intractable in the presence of shallow pointers. E.g., *boolean copy constant propagation* is also intractable for shallow pointers. Thus, if we replace our refcount operations with statements that allow the assignment of constant values zero and one to refcounts (via one level of indirection), and allow copying these values, then verifying that such a value is zero at the end of execution becomes intractable (with just shallow pointers).

5 Related Work

The presence of aliasing in programs is often a hurdle in the design of program analysis. For this reason, there have been various studies to find out how aliasing increases the complexities of different analyses [6, 8, 3]. These papers have studied different problems in the presence of shallow and non-shallow pointers: may and must alias analysis [6], flow-sensitive analyses like constant propagation [8], and type-state properties [3]. Our work studies a problem that is different from each of ones discussed before because of the "context-free" nature of the reference-count property.

Apart from the abovementioned work on complexity-theoretic aspects of dealing with aliasing in the context of doing precise (with respect to a given model) verification or analysis, there has been work on various techniques for dealing with aliasing in the context of verification or analysis, with a potential loss in precision, e.g., see [1, 4].

The problem of reference count verification has been studied previously in the context of model checking under the presence of an unbounded number of threads [2]. They use techniques such as temporal-case splitting to isolate a single object (i.e., resource) and a single thread, which results in a sequential abstraction of the original program. Next, predicate abstraction is used to construct a finite model of the program that can be verified using a model checker. Abstraction refinement is used to construct finer-grained models when the current set of predicates do not suffice to prove the property. Thus, [2] addresses the problem in a more general setting, but does not present any complexity result for this problem. Specifically, they do not provide any polynomial time verification algorithm. We study the same problem in a more restricted setting and show that precise verification is possible in polynomial time in this setting.

References

[1] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–22, New York, NY, USA, 2004. ACM.

[2] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In *TACAS*, pages 352–367, 2009.

[3] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Sci. Comput. Program.*, 58(1-2):57–82, 2005.

[4] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM.

[5] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[6] W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, 1991.

[7] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, 2004.

[8] R. Muth and S. K. Debray. On the complexity of flow-sensitive dataflow analyses. In *POPL*, pages 67–80, 2000.