An Object Ownership Inference Algorithm and its Applications

Samuel E. Moelius III
Drexel University
Philadelphia, PA
smoelius@cs.drexel.edu

Amie L. Souter
Drexel University
Philadelphia, PA
souter@cs.drexel.edu

ABSTRACT

Identifying data sharing relationships in a large program can be a difficult and time consuming task. Understanding such relationships is crucial when building and maintaining complex systems. While languages that automatically deallocate objects provide benefits to the programmer, they also discard information about the lifetimes of objects—information that can provide valuable insight into the semantics of the program.

In this paper, we present a new algorithm for automatically identifying data sharing relationships in Java programs. The relationships identified can aid a software engineer in determining how objects interact in a complex system, and thus, provide insight into how changes to the objects affect the system. We demonstrate the algorithm on an example piece of source code. We think that the algorithm shows promise of becoming the basis of a useful tool for software engineers.

1. INTRODUCTION

In 1968, Dijkstra wrote a landmark article entitled "Go To Statement Considered Harmful[7]." This article was about the proliferation of code with unstructured control-flow graphs as a result of an over reliance on the **go to** statement. In place of the **go to** statement, Dijkstra recommended the use of more structured primitives such as **while-repeat**. No doubt, Dijkstra's words were heard—one can hardly argue that the **go to** statement is as widely used today as it was then.

Nonetheless, today we are in a situation comparable to that of 1968. This time, however, the problem is the unstructured use of references as a result of an over reliance on garbage collection.

Since it can often be difficult or impossible to know at compile time when an object should be deallocated, garbage collection can be a very useful tool. However, the problem with relying on garbage collection is that it throws away too much of the structure of the program. In a program with explicit deallocation, the lifetime of an object can be determined by looking at the source code. This is not true in a program that uses garbage collection.

This has led us to ask the following question: Can one take a program written in a language that features garbage collection, and from it identify the cases in which an object may be explicitly deallocated? Having such a capability has several applications:

- In *program understanding*. Understanding how a program utilizes memory constitutes a significant part of understanding how the program works. For example, a programmer may need to understand how changes to an object affect different parts of the program. Having a tool that could statically determine the lifetime of an object would be beneficial in such cases.

- In *compiler design*. One can imagine producing a compiler for a programming language that features garbage collection, such as Java, that embeds instructions in the compiled code to perform explicit deallocations wherever such opportunities exist.

- In *programming language design*. One can imagine designing a programming language where the programmer explicitly annotates the source code with information regarding the lifetimes of the objects. A compiler for this language statically checks that the program's memory utilization satisfies the annotations.

In this paper, we present an algorithm for inferring certain data sharing relationships in Java programs. At the core of the algorithm is an analysis comparable to escape analysis[2, 5, 3, 16, 15, 11, 4, 6]. The results of the analysis are used to produce an annotated source program that allows one to better understand the program's behavior.

Our algorithm identifies references as *owning*, *borrowing*, or *sharing*. An object whose references are owning and borrowing may be explicitly deallocated. An object whose references are sharing must be garbage collected.

The contributions of this paper are the following:

Proceedings of MASPLAS'04
Mid-Atlantic Student Workshop on Programming Languages and Systems
Seton Hall University, April 3, 2004

- A new algorithm for inferring ownership annotations in Java programs. The algorithm is more versatile than previous approaches in that it allows borrowing references to be returned from methods and assigned to object fields[1].
- A means of communicating the results of escape analysis to the user through the use of ownership annotations.

The remainder of this paper is organized as follows. Section 2 provides a motivating example illustrating the importance of understanding data sharing relationships. The data sharing relationships *owning*, *borrowing*, and *sharing* are described in Section 3. Section 4 describes our algorithm for identifying these data sharing relationships and how we annotate the source code to identify such relationships. Applications of this work to program understanding, compiler design, and programming language design are discussed in Section 5. Section 6 describes related work. We conclude with a summary and future work in Section 7.

2. MOTIVATING EXAMPLE

In order to understand large software systems, it is necessary to understand data sharing relationships between the functions and objects of the system. Consider a function that takes an object as an argument. The following are reasonable questions for a programmer to ask:

1. Who is responsible for deallocating an object—the caller or the callee?
2. If it is the responsibility of the caller, when is it safe to deallocate an object—immediately following a call, or after deallocating another object?
3. Is it safe to reuse an object following a call?

Similar questions may be asked in the case where a function returns an object.

A common theme among these questions is that they all involve *ownership*, i.e. who owns an object, and when does ownership of an object change hands? We normally rely upon documentation to answer such questions. However, documentation can be imprecise and prone to errors. Even worse, the documentation may simply not exist. Thus, programmers would benefit from a tool that could help to answer such questions.

Consider, for example, the code in Figure 1. This small program features two classes: `Message` and `Client`, which implement a message for the purpose of sending over a network and a client that uses it. These classes also make use of another class, `Key`, defined elsewhere in the program.

```
1   class Message {
2     Integer _text;
3     boolean _plain;
4     Message(Integer text,boolean plain) {
5       _text = text;
6       _plain = plain;
7     }
8     Message makePlain(Key key) {
9       if (_plain)
10        return this;
11      else
12        return new Message(key.decrypt(_text),true);
13    }
14    Integer getText() {
15      return _text;
16    }
17  }

18  class Client {
19    void run() {
20      Message msg = request();
21      System.out.println(msg.getText());
22    }
23    Message request() {
24      Key key = new Key();
25      send(key);
26      Message msg = receive();
27      return msg.makePlain(key);
28    }
29    ...
30  }
```

Figure 1: A simple Java program.

The `Client` class first allocates a new session key as an instance of `Key`. It then sends this key out over the network to a remote server. The remote server uses the key to encrypt an instance of `Message` which it then sends back to the client. Upon receiving the message, the client decrypts it using the key, and prints the decrypted message to the console.

For security purposes, we might like to know that no persistent references are stored to the instance of `Key` allocated on line 24. For example, we would hope that `msg` does not hold a persistent reference to this object following the call to `makePlain` on line 27. Fortunately, this can be determined by examining the code. After `Key`'s constructor returns on line 24, a reference to the `Key` object flows to the local variable `key` declared on the same line. From there, a reference to the object flows to several other parts of the program, specifically, the argument of the call to `send` on line 25, the argument of the call to `makePlain` on line 27, and the `this` argument of the call to `decrypt` on line 12. However, `key` has the particularly useful property that it outlives all other references to the object allocated on line 24. That is, assuming that `send` and `decrypt` do not store any persistent references to the object, then all references other than `key` will be destroyed at or before the time at which the call to `makePlain` on line 27 returns. Furthermore, when the call to `request` on line 20 returns, we can be sure that there are no more references to the object.

In contrast, consider the `Message` object returned by the call to `receive` on line 26. The parts of the program that may reference this object cannot be precisely determined at compile time. Rather, it will depend upon the value of the `_plain` field of the object. Specifically, if the value of `_plain` is `true`, then the result of the call to `makePlain` on line 27 will be its `this` parameter. On the other hand, if the value of `_plain` is `false`, then the result of the call to

makePlain will be the object allocated on line 12. As such, it cannot be determined at compile time whether the last reference to the object returned by the call to receive will be msg on line 27 or msg on line 20. Moreover, it cannot be determined whether the lifetime of the object returned by the call to receive extends beyond the time at which the call to request on line 20 returns.

A similar line of reasoning establishes that the Integer object referenced by the _text field of the Message object passed into makePlain also does not have a precisely determined lifetime.

As this example illustrates, a seemingly simple program can have complex behavior when one considers the lifetimes of its objects. Furthermore, understanding the lifetimes of the objects can be necessary in understanding the program's behavior. Of course, this information can always be determined by manually inspecting the program. However, being able to automatically identify situations like those described above saves time, and thus, lessens the overall burden of software maintenance.

3. DATA SHARING RELATIONSHIPS

We have identified several data sharing relationships that provide the programmer with important information about the lifetime of an object. Specifically, we identify the references to an object as *owning*, *borrowing*, or *sharing*. The following gives the intuition behind these terms:

- An owning reference is one that outlives all other references to an object, or that flows to a reference that outlives all other references to the object. Under such circumstances, we say that the object is *owned*.
- A borrowing reference is a reference to an owned object that is not itself an owning reference.
- A sharing reference is a reference to an object that is not owned. Under such circumstances, we say that the object is *shared*.

We continue to use Figure 1 to illustrate these concepts. Since the lifetime of the reference key on line 24 exceeds that of all other references to the Key object allocated on the same line, key is an owning reference. The automatic reference returned by the new statement flows to key, and thus, is also an owning reference. All other references to which these flow are borrowing references. These are the this argument of the call to Key's constructor on line 24, the argument of the call to send on line 25, the argument of the call to makePlain on line 27, and the this argument of the call to decrypt on line 12. By comparison, all of the references to the Message object returned by receive on line 26 are sharing references. Similarly, all of the references to the Integer object referenced by this Message object's _text field are also sharing references.

Objects that have owning and borrowing references have precisely determined lifetimes. As such, they may be explicitly deallocated. Objects that have sharing references, do not have clearly discernible lifetimes and must be garbage collected. In general, owned objects will affect fewer parts of the program than shared objects, as the fact that they have determined lifetimes places restrictions upon which references may refer to them.

4. AUTOMATICALLY INFERRING OBJECT OWNERSHIP

In this section, we present our algorithm to automatically identify references as owning, borrowing, or sharing. In Section 4.1, we begin by describing how we annotate a source program to communicate the results of our algorithm. Section 4.2 describes the means by which these annotations are determined. In Section 4.3, we show the results of applying our algorithm to the code in Figure 1. Finally, Section 4.4 describes a couple of possible variations of the algorithm.

4.1 Annotations

Our algorithm gives every reference in the program an annotation. An annotation may be either a concrete annotation or an annotation variable. A concrete annotation is one of owned, borrowed, or shared, and indicates that the reference is owning, borrowing, or sharing, respectively. In this paper, all annotation variables shall consist of an underscore followed by a number (e.g. _1).

A class is given an annotation parameter for each of its fields that is a reference. We write the annotation parameters in angle brackets (<>) immediately following the class name. We write a field's annotation immediately before its type. Additionally, if the types of any of the class's fields require annotation parameters, then the class is given a corresponding annotation parameter for each. These annotation parameters are written in angle brackets following the type of the field. If the type of a field does not require annotation parameters, then empty angle brackets are written following the type of the field. For example, the Message class from Figure 1 is annotated as

```
class Message<_1> {
    _1 Integer<> _text;
    boolean _plain;
    ...
}
```

Every reference in a method is given an annotation, including the return value, formal parameters, this parameter, and local variables. As before, if the types of any of these require annotation parameters, then a corresponding annotation is given to each.

Following the method's formal parameters, we write the this annotation, followed by the type of this applied to its annotation parameters. For example, the run and request methods of Client are annotated as

```
void run() _14 Client<_14,_19> {
  shared Message<shared> msg = request<...>();
  ...
}

_17 Message<_18> request<...>() _19 Client<> {
  owned Key<> key = new owned Key<><...>();
  ...
}
```

The rules for assigning annotations will be further explained in the next section.

4.2 Algorithm

Our algorithm takes as input a set C of class definitions. The algorithm requires that if C includes a class c_1 that inherits from a class c_2, then c_2 is also in C. The algorithm also assumes that there is no class that inherits from a member of C that is not itself a member of C. If this assumption is not valid, the algorithm will still proceed. However, if one were to rerun the algorithm with such classes added to C, then one may get different results.

Let A be the set of concrete annotations: `owned`, `borrowed`, and `shared`. We define a relation $<:$ over A that captures which concrete annotations may flow to others. The relation is defined as follows:

$<:$	owned	borrowed	shared
owned	×	×	
borrowed		×	
shared			×

Let V be the set of annotation variables. The goal of the algorithm is to find a substitution for the elements of V that maximizes the number of objects identified as owned, and that satisfies the following two properties:

- if the reference annotated by v_1 flows to the reference annotated by v_2, then $v_1 <: v_2$
- for any run of the program, for any owned object, as long as there exists at least one reference to the object, there exists exactly one live reference to the object whose annotation is `owned`

The first step is to determine the annotation parameters of each class. We do this by constructing a graph whose vertices are the elements of C, such that there is an edge from c_1 to c_2 for each field in c_1 of type c_2. Each edge in the graph is given a unique annotation parameter. The annotation parameters of a class c are of all of the edges that are reachable from c within the graph. This is determined by propagating the sets of reachable edges through the graph until a fixed point is reached. Since the number of edges in the entire graph is finite, this process is guaranteed to terminate.

The next step is to assign annotations to the references of each method definition. A method definition is given an annotation for each of the following that are references: the return value, the formal parameters, and the `this` parameter. If the types of any of these require annotation parameters, then a corresponding annotation is given to each. Additionally, each of the method definition's statements is visited to determine where additional annotations are needed. The following are some of the relevant cases:

- Null: `null` is rewritten as `null<`v`>`, where v is a new annotation.

```
 1   class Message<_1> {
 2     _1 Integer<> _text;
 3     boolean _plain;
4a     Message(_1 Integer text,boolean plain)
4b         _2 Message<_3> {
 5       _text = text;
 6       _plain = plain;
 7     }
8a     _4 Message<_5> makePlain(_6 Key<> key)
8b         _7 Message<_8> {
 9       if (_plain)
10         return this;
11       else
12a        return new _9 Message<_10>(
12b          key.decrypt(_text),true);
13     }
14     _11 Integer<> getText() _12 Message<_13> {
15       return _text;
16     }
17   }

18   class Client<> {
19     void run() _14 Client<> {
20       _15 Message<_16> msg = request();
21       System.out.println(msg.getText());
22     }
23     _17 Message<_18> request() _19 Client<> {
24       _20 Key<> key = new _21 Key<>();
25       send(key);
26       _22 Message<_23> msg = receive();
27       return msg.makePlain(key);
28     }
29     ...
30   }
```

Figure 2: The Java program from Figure 1 after all references have been assigned annotations.

- Cast: `(`c`)`e is rewritten as `(`v c`<`$v_1..v_n$`>)`e, where v is the annotation of the expression e, and there is one v_i for each annotation parameter of c. If this is an upcast, then $v_1..v_n$ will be a subset of those associated with the type of e. If this is a downcast, then some of $v_1..v_n$ will be new annotations.
- Object creation: `new` c`(`$e_1..e_n$`)` is rewritten as `new` v c`<`$v_1..v_n$`>(`$e_1..e_n$`)`, where $v, v_1..v_n$ are new annotations.
- Local variable declaration: c x`;` is rewritten as v c`<`$v_1..v_n$`>` x`;`, where $v, v_1..v_n$ are new annotations.

Figure 2 illustrates an annotated version of the code from Figure 1 after all references have been assigned annotations.

As the method definitions are annotated, we record for each annotation the method definition in which it was generated. Let M be the set of all method definitions of all classes in C. The function $\textsc{Method} : V \rightarrow M$ is defined such that if v is the annotation of a reference in m, then $\textsc{Method}(v) = m$.

Once all statements have been annotated, we can begin to infer constraints. For each method definition, a constraint $v_1 <: v_2$ is inferred if there is possible flow of values from the reference annotated by v_1 to the reference annotated by v_2. Additional constraints are inferred when the types

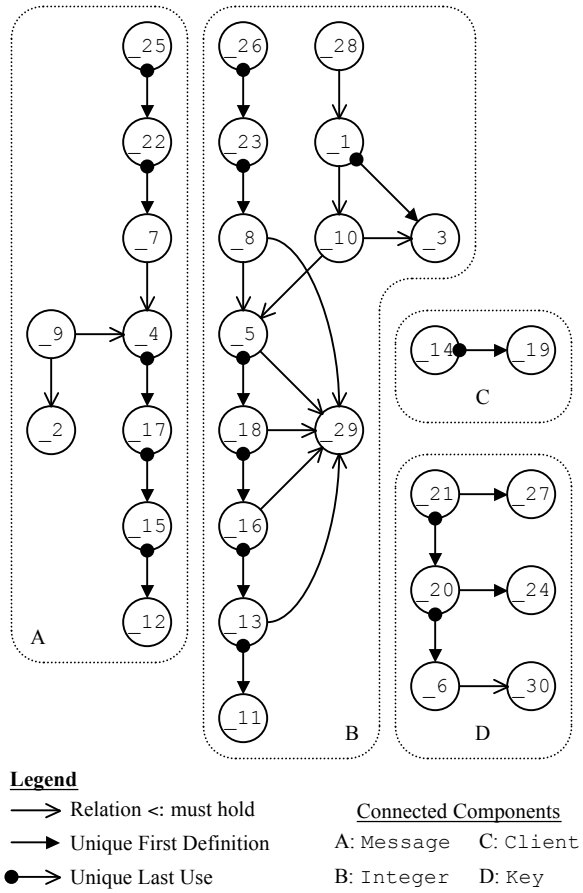

Figure 3: The constraint graph for the `run` method.

of the references annotated by v_1 and v_2 have annotation parameters.

For each constraint, we record whether it was generated by a Unique First Definition (UFD), a Unique Last Use (ULU), or both. A definition of a reference is *first* if it occurs before all other definitions of the reference. A use of a reference is *last* if it occurs after all other uses of the reference. A definition or use is *unique* if it occurs along all control paths. Note that the definition of a formal parameter is always a first definition, while the use of a return value is always a last use.

Once constraints have been inferred for all method definitions, the constraints of each method definition are propagated to its callers. This process continues until a fixed point is reached. Since the number of constraints inferred for the entire program is finite, this process is guaranteed to terminate.

At this point, each method definition has associated with it a set of constraints. This set of constraints naturally forms a graph, which we call the *constraint graph*. In this graph, the vertices are the elements of V, such that there is an edge from v_1 to v_2 if the constraint $v_1 <: v_2$ was inferred.

Figure 3 shows the constraint graph for the `run` method of Figures 1 and 2. For example, the local variable `key` on line 24 of Figure 2 was given the annotation `_20`. Additionally, the reference returned by the `new` statement on the same line was given the annotation `_21`. Since the reference returned by the `new` statement flows to `key`, `_21` <: `_20` was inferred. This fact is reflected in the constraint graph by the edge from `_21` to `_20`.

In order to make statements about the lifetimes of references within the program, we define some additional relations. For $m_1, m_2 \in M$, let $m_1 \rightarrow m_2$ be the relation that says m_1 calls m_2. Equivalently, this relation says that in the call graph, there is an edge from m_1 to m_2. Let $\rightarrow^+$ and $\rightarrow^*$ be the transitive and reflexive-transitive closures of this relation, respectively. Define $\succcurlyeq$ over M and over V as follows:

$$m_1 \succcurlyeq m_2 \quad \stackrel{\text{def}}{=} \quad (m_1 \rightarrow^* m_2) \wedge \neg(m_2 \rightarrow^+ m_1)$$

$$v_1 \succcurlyeq v_2 \quad \stackrel{\text{def}}{=} \quad \textsc{Method}(v_1) \succcurlyeq \textsc{Method}(v_2)$$

In words, $m_1 \succcurlyeq m_2$ says that in the call graph, there is a path, possibly empty, from m_1 to m_2, but there is no non-empty path from m_2 to m_1. This implies that the local variables of m_2 will be destroyed at or before the time at which those of m_1 are destroyed. Thus, $v_1 \succcurlyeq v_2$ says that the reference annotated by v_2 will be destroyed at or before the time at which the reference annotated by v_1 is destroyed.

Let $<:^*$ denote the reflexive-transitive closure of the $<:$ relation. With these relations so defined, we may define the following predicate over annotation variables:

$$\textsc{Final}(v) \quad \stackrel{\text{def}}{=} \quad \forall v' \in V.\ (v <:^* v') \ \Rightarrow\ (v \succcurlyeq v')$$

In words, an annotation variable is FINAL if it outlives all annotation variables reachable from it within the constraint graph. In addition, we define the predicate REACHESFINAL to hold for any annotation variable from which there is a path within the constraint graph to a FINAL annotation variable, such that every edge in the path is both UFD and ULU.

We assign concrete annotations to the elements of V by running the ASSIGNCONCRETEANNOTATIONS procedure from Figure 4 on each method definition. If an annotation variable cannot be assigned a concrete annotation, then the variable becomes an annotation parameter of the method.

4.3 Example Revisited

Figure 5 shows the result of running our algorithm on the code in Figure 1. Recall that our original concern was whether `msg` could hold a persistent reference to the `Key` object allocated on line 24 following the call to `makePlain` on lines 27a-b. Here we see that the local variable `key` on line 24 has been annotated `owned`, indicating that the object it references has a precisely determined lifetime. Within `request`, the only other references to which the reference `key` flows are the argument of `send` on line 25 and the argument

AssignConcreteAnnotations(m)

for each connected component in m's constraint graph

- if the connected component contains the annotation of m's return value, the annotation of one of its formal parameters, or the annotation of its `this` parameter, then leave the connected component unmarked
- otherwise, if for all roots v of the connected component, ReachesFinal(v), then
  (a) for each root v of the connected component, MarkOwned(v)
  (b) for each vertex v' in the connected component that is unmarked, mark v' `borrowed`
- otherwise, mark all vertices in the connected component `shared`

MarkOwned(v)

1. mark v `owned`
2. if there exists an edge from v to v' that is both UFD and ULU and ReachesFinal(v'), then
   - if not Final(v), then MarkOwned(v')
   - otherwise, if Method(v) = Method(v'), then MarkOwned(v')

Figure 4: Pseudocode for assigning concrete annotations to a method definition.

of `makePlain` on lines 27a-b. In both cases, the annotations of these references are `borrowed`. Thus, we can be sure that when the call to `request` on lines 20a-d returns, there are no more references to the object allocated on line 24.

Since it was determined that the `Message` object returned by `request` on lines 27a-b does not have a clearly discernible lifetime, we might expect that the annotation of the return value of `request` on line 23a would be `shared`, but in fact, this is not the case. Instead, the annotation is `_17`, an annotation parameter. This is because constraints were inferred between the annotation of the return value of `request` and the annotation of the return value of `receive`. In effect, this forces `request` to defer the decision as to what concrete annotation should accompany its return value to its caller. Unsurprisingly, if we look at the annotation that is assigned to `_17` in the call to `request` on lines 20a-d, we see that it is `shared`.

4.4 Variations

There are two ways in which the algorithm could be modified that we wish to mention here. These modifications will have relevance to the next section.

The first modification involves reducing the number of annotation parameters inferred for methods. Consider the method `request` on lines 23a-28 of Figure 5. Of the 17 annotation parameters inferred for this method, 16 involve `Message` and `Integer` objects. These 16 annotation parameters are all highly interdependent. In other words, there are fewer than 3^{16} possible assignments of these parameters. Thus, one could reduce the number of annotation parameters by identifying annotation parameters that have the same concrete annotation under all valid assignments. Two such annotation parameters could be replaced by one.

```
 1   class Message<_1> {
 2     _1 Integer<> _text;
 3     boolean _plain;
4a     Message<_1,_2,_3>(_1 Integer text,boolean plain)
4b         _2 Message<_3> {
 5       _text = text;
 6       _plain = plain;
 7     }
8a     _4 Message<_5> makePlain<_1,_2,_3,_4,_5,_6,_7,
8b         _8,_9,_10,_28,_29,_30>(_6 Key<> key)
8c         _7 Message<_8> {
 9       if (_plain)
10         return this;
11       else
12a        return new _9 Message<_10><_1,_2,_3>(
12b          key.decrypt<_28,_29,_30>(_text),true);
13     }
14a    _11 Integer<> getText<_11,_12,_13>()
14b        _12 Message<_13> {
15       return _text;
16     }
17   }

18   class Client<> {
19     void run() _14 Client<_14,_19> {
20a      shared Message<shared> msg = request<shared,
20b        shared,shared,shared,shared,shared,shared,
20c        shared,shared,shared,_19,shared,shared,
20d        shared,shared,shared,shared>();
21a      System.out.println<>(msg.getText<shared,
21b        shared,shared>());
22     }
23a    _17 Message<_18> request<_1,_2,_4,_5,_7,_8,_9
23b        _10,_17,_18,_19,_22,_23,_25,_26,_28,
23c        _29>() _19 Client<> {
24       owned Key<> key = new owned Key<><borrowed>();
25       send<borrowed>(key);
26       _22 Message<_23> msg = receive<_25,_26>();
27a      return msg.makePlain<_1,_2,_4,_5,borrowed,_7,
27b        _8,_9,_10,_28,_29,borrowed>(key);
28     }
29     ...
30   }
```

Figure 5: The Java program from Figures 1 and 2 after concrete annotations have been assigned.

Another way to reduce the number of annotation parameters would be to determine whether an annotation parameter was assigned the same concrete annotation in all uses of the method. In such a case, the annotation parameter could simply be eliminated, and all of its uses replaced with the concrete annotation.

The second possible modification to the algorithm involves handling cases where there is more than one concrete annotation that may be assigned to an annotation variable. This can occur, for example, when the ULU of an owning reference annotated by v_1 is also the UFD of a reference annotated by v_2, and v_1 outlives v_2. In such cases, it would be equally valid to assign either `owned` or `borrowed` to v_2.

To put it another way, it is possible for there to be an edge

in the constraint graph from v_1 to v_2 that is both UFD and ULU, yet both v_1 and v_2 can be FINAL. The following are a few strategies for handling such cases:

1. Never mark v_2 `owned`
2. Mark v_2 `owned` only if $\text{METHOD}(v_1) = \text{METHOD}(v_2)$
3. Always mark v_2 `owned`

These strategies are ordered such that the first assigns `owned` to as few annotation variables as possible, while the third assigns `owned` to as many annotation variables as possible. The algorithm as we have presented it in Figure 4 implements the second of these three strategies. Essentially, this strategy assigns `owned` to as many of the v_2 of the same method as v_1 as possible, but does not assign `owned` to the annotation variables of called methods unless v_1 was not FINAL to begin with. It is not clear, however, that this is a *best* strategy. We will say more about this in the next section.

5. APPLICATIONS

In the introduction, we mentioned three areas in which our algorithm has applications: program understanding, compiler design, and programming language design. We will now address each of these areas individually.

5.1 Program Understanding

As our example of Figures 1, 2, and 5 illustrates, understanding how a program utilizes memory constitutes a significant part of understanding how the program works. As such, our algorithm could be used as the basis of a tool for program understanding.

The most straightforward use would be to construct a tool that simply outputs the annotated source code produced by the algorithm. In fact, we have a prototype implementation built upon the Eclipse framework that does this for a subset of Java. However, one could argue that the annotated source code contains too much information to be useful to humans. Specifically, the number of annotation parameters inferred for methods might be more than one would like.

One could apply one of the modifications described in the previous section to reduce the number of annotation parameters inferred. However, we feel that a better approach would be to leave the algorithm unmodified, and instead construct a tool that *interprets* the results for the user. Specifically, we envision a tool that communicates the results of the algorithm to the user through syntax highlighting. Such a tool might perform the following functions:

- Display each reference in one of four colors, depending on whether its annotation is `owned`, `borrowed`, `shared`, or an annotation parameter.
- Allow the user to select an owning reference and identify the chain of owning references to and from which the reference flows.
- Allow the user to select an owning reference and identify the tree of borrowing references to which the reference flows.

As mentioned previously, there are cases where there will be more than one concrete annotation that may be assigned to an annotation variable, and there are different strategies for doing so. It is not clear to us that any single strategy is best suited to program understanding. That is, we are of the opinion that different strategies will be better at illustrating a program's behavior in different situations. As such, we believe that a tool would be most useful by allowing the user to select from several strategies with which to run the algorithm on a piece of source code and compare the results.

5.2 Compiler Design

If in a program one can separate out the set of objects that may be explicitly deallocated, then one can reduce the load of garbage collector, thereby obtaining improved performance. Our algorithm could be used as the basis of such an optimization in an optimizing compiler.

Objects that our algorithm identifies as owned may be explicitly deallocated. For every such object, there will be an owning reference that does not flow to any other owning reference. When this reference leaves scope, the object to which it refers may be deallocated.

It may be the case that such a reference is annotated by an annotation parameter of the method in which it occurs. Thus, it may be the case that the object referred to should be deallocated in some calls of the method, but not in others.

There are several ways in which this could be implemented. One way would be to have multiple versions of the method—one for each possible assignment of the method's annotation parameters. This solution would be analogous to the way in which C++ handles templates. Since in each version of the method there would be no annotation parameters, the objects that should be deallocated could be precisely determined. Instructions would then be embedded in the compiled code to perform the deallocations where appropriate.

This solution would likely be fast, but could also have a drastic effect on the size of the compiled code. In the worst case, the increase would be exponential on the number of annotation parameters. To mitigate this effect, one would first have to reduce the number of annotation parameters of each method using one of the modifications described in the previous section.

A second approach to implementing this idea would be to encode which objects should be deallocated in an additional formal parameter of the method. This additional formal parameter would be a bit string with one bit for each reference in the method. If in a particular call to the method the object referred to by a reference should be deallocated, then that reference's bit would be set. The bit would be left unset, otherwise. Instructions would then be embedded in the compiled code to check this bit when the reference leaves scope, and perform the deallocation if the bit is set.

This solution would avoid the potential increase in code size associated with the previous solution, but would likely not be as fast due to the additional runtime checks that would be necessary. Nonetheless, the overhead associated with these runtime checks would likely be less than that if the objects

were garbage collected. Also, to increase the likelihood that the additional formal parameter would fit into a single word, one would again have to reduce the number of annotation parameters of each method using one of the modifications described in the previous section.

5.3 Programming Language Design

There is no doubt that in many cases a programmer will have an idea as to when the last reference to an object should leave scope, and thus, when references to the object should no longer exist. Additionally, if a programmer were using an optimizing compiler like that described previously, it is likely that there will be cases where the programmer expects the optimization to be performed. Thus, it seems natural to want to allow the programmer to place annotations in the source code, and to alert him when the annotation inferred does not match that which he intended. Essentially, what we are describing is type system for a programming language.

In most type systems, the type of a value captures all of the information that we would wish to know about the value that can be obtained at compile time. However, if we want this to be true of the type system that we are describing, then the type of a method must incorporate more information than just that which is currently included in the annotated source code. More specifically, the concrete annotations that may be assigned to a method's annotation parameters are dependent upon the method's definition. If the system had to reanalyze the method's definition at each use of the method, this would somewhat defeat the purpose of having a type system. Thus, the type of a method must incorporate more information than just its annotation parameters and the annotations of its return value, formal parameters, and `this` parameter—it must also encode the constraint graph. That is, the type a method must include the annotation variables for which $<:$ must hold, and when this condition arose as the result of a UFD or ULU.

6. RELATED WORK

Our work was largely inspired by that of Aldrich, *et. al.*[1]. In [1], Aldrich, *et. al.* define a language called AliasJava and develop an algorithm to infer AliasJava programs from Java programs. AliasJava is very similar to the annotated version of Java that we present in this paper. However, there are important differences between the two systems, both conceptual and technical.

The focus of AliasJava is on determining which objects may reference which others. Thus, ownership in AliasJava is a statement about the relationship between two objects.

Our focus, on the other hand, is on determining which references may refer to which objects. Thus, ownership in our system is statement about the relationship between a reference and an object.

Nonetheless, AliasJava does have idioms for representing owning and borrowing references.[1] However, our approach is more versatile than that of AliasJava. Specifically, AliasJava does not allow borrowing references to be returned from methods or assigned to object fields. For our purposes, this restriction would disallow too many valid programs. Consider, for example, the `min` function defined in Figure 6. This function takes two `Integer` objects as arguments, determines the lesser of the two, and returns that object. This function can be used in a manner entirely consistent with notion that borrowing references should not outlive owning references. Nonetheless, this function would not be allowed in AliasJava simply because it returns a borrowing reference.

```
borrowed Integer min(borrowed Integer x,
        borrowed Integer y) {
    if (x.intValue<borrowed>()
            <= y.intValue<borrowed>())
        return x;
    else
        return y;
}
```

Figure 6: A function that returns a borrowing reference.

An AliasJava program that uses no nonstandard idioms other than owning, borrowing, and sharing references will be a valid program in our system. However, our algorithm will identify objects as owned that could not be so identified in AliasJava.

The core of our algorithm is comparable to escape analysis[2, 5, 3, 16, 15, 11, 4, 6]. Escape analysis is the process of determining the set of objects that may be referenced from outside of a given function. Such objects are said to *escape* the function. All other objects are said to be *captured* by the function. Typically, escape analysis involves some form of *may-alias* analysis where the results of the analysis are reflected in a graph.

A typical application of escape analysis is in identifying cases where optimizations such as synchronization elimination and stack allocation may be performed. In such an application, it always safe to identify an object as escaping—if an object is incorrectly identified as escaping, the optimization is simply not performed. As such, a may-alias analysis is typically enough. That is, if an object *may* be referenced from outside of a function, then the object *may* escape the function.

However, our algorithm also needs to know when an object *must* escape a function. As such, our analysis is also a *must-alias* analysis. Specifically, when an owning reference to an object leaves scope and borrowing references to the object still exist, then the owning reference *must* flow to to another owning reference. We identify this as the case when a flow of values occurs as the result of a UFD and ULU.

Region inference is another approach to solving problems similar to those that we address[13, 12, 10, 9, 14]. Intuitively, a region is a set of objects. Regions are explicitly allocated and deallocated. When one allocates an object, one specifies the region in which to allocate the object. Deallocating a region deallocates all of the objects inside of it.

[1]Although, their terminology is slightly different than ours. Whereas we use the terms *owning* and *borrowing*, they use the terms *unique* and *lent*, respectively.

The goal of region inference is to embed code within a program to explicitly allocate and deallocate regions, and to allocate the objects of the program within those regions appropriately.

Region inference alone is not enough. That is, the memory consumption of a program to which region inference has been applied may be unbounded at runtime, as demonstrated in [9]. The problem arises from objects that must be allocated in a top-level region that is not deallocated until the program terminates.

There has been recent work on combining region inference and garbage collection[9, 8]. This is useful for the compiler writer. However, in regard to program understanding and programming language design, it is not clear that regions are the best means of communicating the lifetimes of objects to the programmer.

There has also been recent work on applying regions to Java[10]. However, thus far, this has been in the form of runtime optimizations, not compile time analyses.

7. CONCLUSION AND FUTURE WORK

In this paper, we present an algorithm for automatically identifying data sharing relationships involving ownership in Java programs. The algorithm is more versatile than previous approaches in that it allows borrowing references to be returned from methods and assigned to object fields. It is also, to our knowledge, the first attempt to apply escape analysis to program understanding. The algorithm also has applications to compiler design and programming language design.

We are presently working on a formal description of the algorithm, and a proof of its correctness. In addition, we are are working on an implementation of the algorithm that handles all of Java. With such an implementation, we will be in a position to better evaluate the usefulness of the algorithm on large programs.

8. REFERENCES

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330. ACM Press, 2002.

[2] Bruno Blanchet. Escape analysis: correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37. ACM Press, 1998.

[3] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 20–34. ACM Press, 1999.

[4] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, 2003.

[5] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM Press, 1999.

[6] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[7] Edsger W. Dijkstra. Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[8] Martin Elsman. Garbage collection safety for region-based memory management. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 123–134. ACM Press, 2003.

[9] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 141–152. ACM Press, 2002.

[10] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for java. In *Proceedings of the third international symposium on Memory management*, pages 127–138. ACM Press, 2002.

[11] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 12–23. ACM Press, 2001.

[12] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[13] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 188–201. ACM Press, 1994.

[14] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[15] Frederic Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2001.

[16] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 187–206. ACM Press, 1999.