Interactive Ownership Type Inference

CMU SCS Senior Thesis

Will Cooper

Advisor: Jonathan Aldrich

Abstract

Ownership types have the potential to strengthen encapsulation in object-oriented programming languages. However, annotating existing code can be tedious, and fully automated ownership type inference has not worked well enough, with the result that it is difficult to obtain the benefits of ownership types in practice.

We address this problem with an interactive ownership type inference tool that obtains information from the programmer about intended encapsulation properties and uses this to infer ownership types. This will allow developers to quickly annotate existing code, enabling them to obtain the benefits of ownership types.

1. Introduction/Motivation

Object-oriented languages use access modifiers to help enforce encapsulation. However, an object's internal representation can be exposed even without violating access modifiers; for example, in Java 1.1 a security function returned a pointer to an internal array rather than a copy, potentially allowing a malicious applet to modify the array to pose as trusted code. Ownership types help to strengthen encapsulation by controlling aliasing (multiple references to the same object).

This paper describes an improvement in the usability of the ownership type system AliasJava [1, 2], although we believe the concepts could be applied to other ownership type systems as well. AliasJava uses `unique` to indicate an unaliased reference; `owned` to indicate a reference that may not be aliased by any reference not owned by the same object; ownership parameters to grant objects access to references owned by other objects; `shared` to indicate a reference with unrestricted aliasing; and `lent` to indicate a temporary, “borrowed” reference. See below for more details.

An inference tool was created for AliasJava to facilitate annotating large pre-existing libraries, which would be tedious to annotate by hand. However, two problems appeared when the system was tested. First, there was no way to obtain an explanation for why a particular annotation was inferred, so if an inferred annotation did not match the intended behavior, it would be very difficult to debug. Second, far too many parameters were inferred for many classes (hundreds in some cases); the tool seemed to be making distinctions between parameters that are too fine to be useful to the programmer.

The main issue seems to be that programmer intent cannot be inferred; an automated inference tool cannot know how a variable "should be" annotated, and it cannot know when two parameters are similar enough that it would not be useful to keep them separate.

In this paper we present a solution to these problems by making inference interactive: allow more programmer input to the inference process, and provide feedback to the programmer on why annotations were inferred. The interactive inference tool is based on the original fully automatic inference tool; we have modified it to handle ownership type annotations in the source code and to provide certain kinds of feedback to the programmer.

2. Overview of AliasJava

This overview is based heavily on [1].

a. Annotations Used in AliasJava

As stated above, AliasJava uses five annotations: `unique`, `owned`, ownership parameters, `shared`, and `lent`.

Unique

A newly created object is *unique*; that is, there is only one reference to the object. We annotate a type with `unique` to describe a reference that does not have persistent aliases.

In general, after a `unique` variable or field is read, the source location must be dead (that is, unused by subsequent code); otherwise the read reference would be an alias of the supposedly unique source.

In AliasJava, `unique` can be considered a universal source: `unique` values can be assigned to a location with any other data sharing annotation, as long as the original reference is destroyed afterwards. The converse is not true, as the other data sharing annotations cannot guarantee that a value is unique.

Owned

An object may need to maintain invariants over its state that could be violated if some of its fields have external aliases. Making these fields `private` is not sufficient because references to the objects could still escape the class as, for example, the return value of a `public` method.

The `owned` annotation describes a reference that is confined to the scope of the enclosing object, unless that object explicitly gives another object permission to access it. Owned references may only flow to `owned` variables within the scope of the enclosing object.

Ownership Parameters

An object may need to structure its representation by putting some of its objects into a container that is also part of its representation. In this case, we can pass `owned` as an *ownership parameter* to the container class, granting that class the capability to reference the element data that are owned by another object.

Shared

Some objects may need to be shared throughout a program, and thus cannot be confined by an owning object. We give references to such objects a `shared` annotation, representing the fact that these objects may be shared globally. Unfortunately, little reasoning can be done about `shared` references, except that they may not alias non-shared references. However, `shared` references are essential for interoperating with existing run-time libraries, legacy code, and `static` fields, all of which may refer to aliases that are not confined to the scope of any object instance.

Lent

Suppose that we want to compare two `unique` objects with, for example, the `equals` or `compareTo` methods. In order to do this with the annotations so far, we would have to destroy our reference to one of the objects so that it can flow to the method argument. Instead, we can annotate the method argument as `lent` to indicate that it is a temporary alias. A `unique` object can be passed to a method as a `lent` argument even without destroying the original `unique` reference. The method can pass on the object as a `lent` argument to other methods, but cannot return it or store it in a field. Thus, the `lent` annotation preserves all of the reasoning about the unique object, but adds a large amount of practical expressiveness. The `lent` type can also be used to temporarily pass an `owned` object to an external method

for the duration of a method call, without any risk that the outside component might keep a reference to that object. Therefore, `lent` can be considered a universal sink: values with any alias type annotation may be assigned to a `lent` location. The converse is prohibited: `lent` values may only be assigned to other `lent` locations.

b. An Example of Usage

Figure 1 illustrates the use of ownership parameters. The `StackClient` class uses a `Stack` to hold integers that are part of its representation. When the `StackClient` creates a `Stack`, it passes the `owned` capability as the `Stack`'s parameter to give the `Stack` permission to access the objects owned by `StackClient`. The code in `run` shows that `Integers` owned by the `StackClient` can be pushed onto and popped off the stack.

```
public class StackClient {
  unique Stack<owned> st=new Stack<owned>();

  public void run() {
    owned Integer i = new Integer(5);
    st.push(i);
    owned Integer i2 = (Integer) st.pop();
  }
}

public class Stack<element> {
  private owned Link<element, owned> top;

  public element Object pop() {
    if (top == null)
      return null;
    owned Link<element, owned> temp = top;
    top = top.next();
    return temp.get();
  }
  public void push(element Object o) {
    top = new Link<element, owned>(o,top);
  }
}

public class Link<element, link> {
  private link Link<element, link> nxt;
  private element Object obj;

  public Link(element Object _obj,
              link Link<element, link> _nxt) {
    obj = _obj; nxt = _nxt;
  }
  public element Object get() {
    return obj;
  }
  public link Link<element, link> next() {
    return nxt;
  }
}
```

Figure 1. A `Stack` class parameterized by the owner of its elements, a `Link` class used in the stack's representation, and a client of the stack.

The stack uses a linked list to store its elements. References to the links in the list should be confined to the enclosing `Stack` object, and so the head of the list (that is, the top of the stack) is annotated `owned`. Since the linked list is a recursive data structure, each link is parameterized with a capability to access not only the elements of the list (owned by the `StackClient` in this example), but

also the other links in the list (owned by the `Stack`). Therefore, the `Stack` passes the `owned` capability as the second parameter of the links in the linked list.

c. Properties Ensured by AliasJava

AliasJava ensures uniqueness and ownership invariants that restrict the aliasing patterns that can occur during program execution.

The uniqueness invariant states the obvious fact that variables and fields with the `unique` annotation hold unique references.

> *Uniqueness Invariant:* At a particular point in dynamic program execution, if a variable or field that refers to an object `o` is annotated `unique`, then no other field in the program refers to `o`, and all other local variables that refer to `o` are annotated `lent`.

The ownership invariant states that ownership annotations are consistent across program variables and across program execution.

> *Ownership Invariant:* At a particular point in dynamic program execution, if a variable or field referring to object `o` has an ownership annotation denoting object `o'`, then all other variables or fields that refer to `o` at any subsequent point in dynamic program execution, are either annotated `lent` or have an ownership annotation denoting the same owner `o'`.

Another way to state the ownership invariant is that each non-`unique`, non-`shared` object is owned by exactly one other object. Only an object's owner, and the objects that the owner has delegated a capability to, may store a reference to that object.

3. Using Interactive Ownership Type Inference

a. Programmer Input

The basic idea of how to use the interactive inference tool is fairly simple: annotate none, some, or all of the variables in a set of Java classes with ownership types, and then run the tool on the .java files.

There are three kinds of annotations that can be put in the source code: formal parameters of classes and interfaces, and alias types and actual parameters of variables (including fields and the arguments and return values of methods). These are all specified using standard AliasJava syntax where user annotations are desired, and may be left out anywhere that they are not desired. There are a few noteworthy points regarding each:

Formal Parameters: If the programmer wants the class's inferred parameter list to not be any longer than the given parameter list, this can be specified by adding "complete" to the beginning of the parameter list. For example, to specify that a class must have no parameters, provide the parameter list "<complete>". (This is admittedly a rather inelegant implementation, but it does work.) Note that only the number of parameters is stored, not their names; if any user-specified formal parameters do not appear in the code, then they may be absent from the inferred formal parameter list, or they may be replaced by automatically generated parameter names.

Alias Types: Any identifier may be used as an alias type. If an alias type other than `lent`, `shared`, `owned`, or `unique` is specified, it is assumed to be a parameter of the nearest enclosing class whose user-specified list of formal parameters contains that word, or the class directly containing the variable if the word does not appear as a formal parameter of any enclosing class. (Thus, unless the list is specified as being complete, the programmer need

only provide those formal parameters which will correspond at some point to a user-specified actual parameter.)

Actual Parameters: If a list of actual parameters is provided for a variable, then the actual parameters are matched to the formal parameters of the variable's type in order. As with alias types, any identifier may be used. The list of actual parameters may not be longer than the type's user-specified list of formal parameters, but it may be shorter, in which case any formal parameters beyond the number of actual parameters are left unspecified. If no actual parameter list is provided, all parameters are left unspecified. Particular parameters may also be left unspecified by using the special keyword `any`.

Additionally, there is one new kind of input to the inference tool other than annotations. We have discovered that it is sometimes necessary to merge two nodes in the "constraints graph" used in inference if a certain kind of path between them exists. These paths can in theory be arbitrarily long; however, we have found that in practice it is sufficient to only inspect relatively short paths. The maximum search depth (i.e. half the length of the longest path to inspect) can be specified by a command-line argument; otherwise a default value is assumed. If the tool reports an error on a valid program, then the path-checking algorithm is probably not searching deep enough, so the tool should run to completion if the search depth is increased sufficiently.

b. Feedback

The tool is able to provide two kinds of feedback: error reporting and inference tracing.

<u>I. Error Reporting</u>

Whenever programmers can provide types, there is the possibility that the types are invalid. The tool's response to invalid types depends on what types are invalid.

If a variable has a user lent or unique annotation and the tool determines that the variable cannot be `lent` or cannot be `unique` according to the normal rules of AliasJava, then it nevertheless assumes that the user annotation is "correct" in some sense. The motivation for this behavior is that we have found that some aliases can be considered harmless. For example, the argument to the `equals` method of the `Object` class was not inferred to be `lent` by the original inference tool because under certain circumstances it can be used as a synchronization key; but since a hash code could be used as this key instead of the object without changing the functionality, it is more or less irrelevant that this alias exists, so the argument to `Object.equals` still matches the conceptual intention of the `lent` type. Although inference continues as if invalid user `lent` and `unique` annotations are correct, the invalid annotations are reported to the user after inference finishes, and information about inconsistencies can be requested during inference tracing.

If the tool decides that two variables must and must not have the same `owned`, `shared`, or parameter annotation, then it terminates, and if possible reports which two variables caused the error. Unfortunately, the inference algorithm for `owned`, `shared`, and parameter annotations is complex, so it is not always possible to determine exactly what caused the error; even when two variables can be named, the names do not always provide much information if the programmer cannot figure out why those variables were inferred to have the same and different annotations.

The final kind of error that might occur happens when a class has a user-specified complete formal parameter list and the inference tool cannot reduce the class's inferred parameter list to fit the user-specified list. In this case, the tool terminates and reports which type's parameter list it was unable to sufficiently reduce.

II. Inference Tracing

Inference tracing allows the user to ask why variables cannot be `lent` or cannot be `unique`. It is only performed if the user requests it, which can be done via a command-line argument or at a prompt if invalid user `lent` or `unique` annotations are reported.

The user may request lentness, uniqueness, or consistency feedback for any variable in the files on which the tool was just run. For lentness feedback, if the query variable is `lent` or is base-case non-`lent` (e.g. a field), then the tool simply states this. Otherwise, it provides a list of base-case non-`lent` variables to which the query variable transitively flows; the user may then request the shortest flow path to any of these base cases. Uniqueness feedback is similar, but the base cases are different and the relevant flow is in the opposite direction (the tool reports base cases which flow to the query variable). For consistency feedback, if the query variable is not user-annotated as `lent` or `unique` or if its annotation is valid, then the tool simply states this; otherwise, feedback is the same as for lentness or uniqueness depending on which annotation the variable has.

The user can also request that the tool simulate the removal of a flow constraint; this is useful for finding out how the code would have to be changed to remove inconsistencies. The flow graph can be restored to its original state after constraints are removed if the user wants this.

We have not figured out any similarly useful feedback that could be provided about inference of other annotations. The inference algorithm for `shared`, `owned`, and parameters involves manipulating a graph according to a set of rules; the chain of events linking the source code to a particular change in the graph can be quite long and complex, and any attempt to keep track of all of it would increase the tool's memory requirements and possibly its running time enough to make it impractical.

Inference Tracing Example

When the original inference tool was run on a large subset of an older version of the Java standard library, the argument to `Object.equals` was not inferred to be `lent`. Using inference tracing, we were able to figure out why. First, we ran the interactive inference tool on the java.lang and java.util packages of the Java 1.4 standard library, with the inference tracing option selected. When the tool finished inferring and reached the inference tracing prompt, we entered the following to ask whether the argument to `equals` is `lent`, and if not, then why not:

```
-> l:java.lang.Object.equals(Object).obj
```

The tool produced a list of sources of the non-lent annotation, which happened to consist of only one field:

```
Sources of the non-lent annotation of java.lang.Object.equals(Object).obj
(from shortest to longest trace):
0: java.util.Collections.SynchronizedCollection.mutex
```

To learn how the argument flows to this field, we entered its index in the above list:

```
-> 0

Trace from node java.lang.Object.equals(Object).obj to source node this.mutex:
0: java.lang.Object.equals(Object).obj
1: java.util.AbstractMap.equals(Object).o
2: java.util.IdentityHashMap.equals(Object).o
3: java.util.IdentityHashMap.equals(Object).if1.else.if1.else.if1.then.m
4: java.util.Map.entrySet().this
5: java.util.Hashtable.entrySet().this
6: java.util.Collections.synchronizedSet(Set,Object).mutex
7: java.util.Collections.SynchronizedSet.<init>(Set,Object).mutex
8: java.util.Collections.SynchronizedCollection.<init>(Collection,Object).mutex
9: java.util.Collections.SynchronizedCollection.mutex
```

Using this flow path as a guide, we inspected the source code and found that the `mutex` is only used to obtain a unique identifier; it is not used in any way that could cause aliasing bugs. Therefore, the `mutex` argument of the `SynchronizedCollection` constructor could be considered `lent` even though it flows to the `mutex` field. To check what would

happen if this flow constraint were removed, we entered the following, which means "make it so that the `lent`ness of node 9 in the above list does not constrain the `lent`ness of node 8":

```
-> -l:9,8
```

To check whether there were any other flow paths to this field, we entered

```
-> l:java.lang.Object.equals(Object).obj
```

again; this time the response was

```
No remaining traces to non-lent nodes
```

We were able to deduce that we could cause the argument to `Object.equals` to be inferred as `lent` simply by annotating the `mutex` argument of the `SynchronizedCollection` constructor as `lent`, or by reworking the code so that the argument is not stored as a field (for example, by changing the field to an `int` and putting the line "`this.mutex = System.identityHashCode(mutex);`" in the constructor).

4. Algorithmic Changes

a. Overview of Original Inference Algorithm

The interactive inference tool is a modified version of the original inference tool. In order to fully understand the algorithms used by the new tool, it is necessary to understand the algorithms used by the old tool. The following is based heavily on [1].

The inference algorithm begins by inferring `lent` annotations, since this annotation is the most general (a value with any other annotation can be assigned to `lent`) and since it can be inferred independently from other annotations. We next infer `unique` annotations using an algorithm that depends only on the inferred `lent` annotations. We infer the remaining annotations in a final pass.

I. Inferring Lent

We infer `lent` annotations with a constraint-based algorithm. Our algorithm assigns either `lent` or `non-lent` to each local variable, expression, and method parameter of reference type, and to the `this` reference for each method. Initially, we optimistically assume that all annotations are `lent`. We then assign `non-lent` annotations the base-case expressions that may not be `lent`: fields and the return values of methods.

Next, the algorithm constructs a directed graph capturing the value flow between the variables and expressions in the program. The final annotations can be computed by traversing this graph backwards from all `non-lent` nodes, so that if an expression `a` flows to expression `b`, and `b` is `non-lent`, then `a` must be `non-lent` as well. Intuitively, this represents the constraint that a `lent` value may not be assigned to a `non-lent` variable. All nodes in the graph that are not backwards reachable from `non-lent` nodes can safely be annotated `lent`.

II. Inferring Unique

The algorithm for inferring `unique` annotations is similar to the `lent` algorithm above. The algorithm assigns either `unique` or `non-unique` to each program variable and expression. As before, we optimistically assume that all annotations are `unique`.

We divide value flow into two cases: ordinary assignments (`x = y`), where both `x` and `y` are live after the assignment, and last assignments ($x =_{last} y$), where `y` is dead after the assignment.

For each ordinary assignment `x = y` we require that `x` is `non-unique`, since it must alias the value y that is not dead. In addition, if `x` is not `lent`, then `y` must also be `non-unique`, since it must alias `x` after the assignment.

The rule for last assignments `x =`$_{last}$ `y` is simple: if `y` is `non-unique`, then `x` must be `non-unique` also. Since `y` is dead after the assignment, if we can prove that `y` was unaliased before the assignment, we know that `x` is unaliased after the assignment. Thus, starting from the `non-unique` base cases generated from ordinary assignments and native methods, we can propagate `non-unique` forward along the directed graph formed by last assignments. All remaining variables and expressions are `unique`.

III. Inferring Other Annotations

In order to infer the remaining alias annotations, we adapt a constraint-based alias analysis that solves equality, component, and instantiation constraints over type variables.

Due to space constraints, we cannot present the full details of the inference algorithm. Instead, we present a high-level overview of the algorithm in parallel with an example that illustrates many of the key issues. We choose as our running example the `Stack` code in Figure 1, assuming initially that none of the alias annotations in that figure is present. Our goal will be to infer the alias annotations given in Figure 1. The discussion below focuses on the core of the inference algorithm, which infers the alias parameters for each class. Later, we will describe how to integrate the other annotations into the constraint-based framework.

Analysis Setup. We begin our analysis by creating a unique node for every variable, method argument or result, class, field, and expression in the program text. This node is a type variable representing the alias annotation for the corresponding declaration or expression. Distinct type variables indicate distinct alias parameters of the enclosing class.

Figure 2(a) shows the type variables generated from Figure 1. For example, the code in the `Stack` class includes the type variables *Stack*, *top*, *pop*, *temp*, and *o* (we abbreviate the type variable for a method result by the method name). To simplify the presentation, we ignore certain anonymous type variables generated from program expressions. Our analysis solves three different forms of constraints: equality, component, and instantiation, which are described in turn below.

(a) Initial variables:

class StackClient:	*StackClient, st, i, i2*
class Stack:	*Stack, top, pop, temp, o*
class Link:	*Link, obj, nxt, _obj, _nxt, get, next*

(b) Initial constraints:

Equality:

$top = temp$	$obj = _obj$	$nxt = _nxt$
$obj = member$	$next = nxt$	

Component:

$StackClient\ _{i}\ i$	$StackClient\ _{i2}\ i2$	$StackClient\ _{st}\ st$
$Stack\ _{top}\ top$	$Stack\ _{pop}\ pop$	$Stack\ _{temp}\ temp$
$Stack\ _{o}\ o$	$Link\ _{obj}\ obj$	$Link\ _{nxt}\ nxt$
$Link\ _{obj}\ _obj$	$Link\ _{_nxt}\ _nxt$	$Link\ _{get}\ get$
$Link\ _{next}\ next$		

Instantiation:

$Stack \leq_{st} st$	$Link \leq_{top} top$	$Link \leq_{temp} temp$
$Link \leq_{nxt} nxt$	$Link \leq_{_nxt} _nxt$	$Link \leq_{next} next$
$o \leq_{st} i$	$pop \leq_{st} i2$	$next \leq_{top} top$
$get \leq_{temp} pop$	$_obj \leq_{top} o$	$_nxt \leq_{top} top$

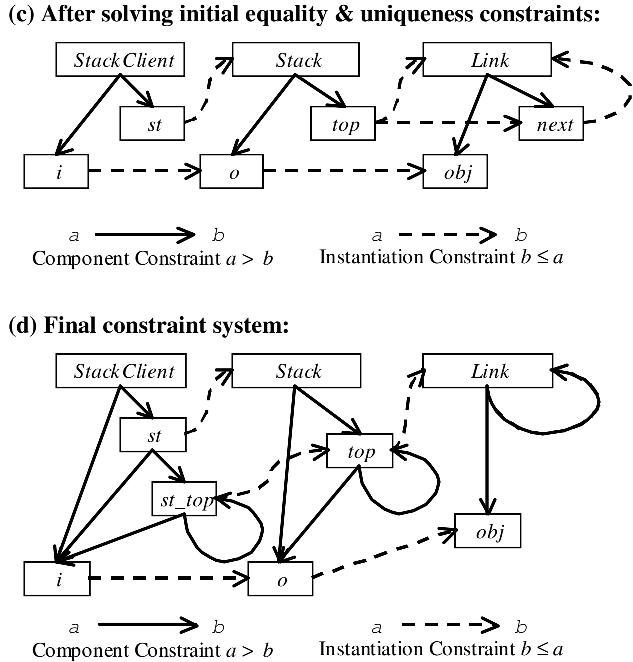

Figure 2. Constraints generated and solved during inference of the alias types given in Figure 1.

Equality Constraints. When a value flows from one variable to another within a class, we generate an equality constraint $a = b$, indicating that the two corresponding type variables

must represent the same alias annotation. For example, our analysis generates the equality constraint *top* = *temp* due to the assignment `temp = top` in line 6 of the definition of `Stack`. However, we do not generate equality constraints for value flow between variables in different classes. For example, even though the method `pop` returns the result of calling `get`, we don't equate the corresponding *pop* and *get* variables, because that would place unnecessary constraints on other parts of the program that use `Link.get`. We use instantiation constraints (discussed below) to reason about value flow between classes in a way that treats different `Link` objects differently. Figure 2(b) shows the equality constraints generated from Figure 1.

In our implementation, equality constraints are solved via unification using a union-find data structure. Thus, for the equality constraint *top* = *temp*, we choose *top* arbitrarily as the equivalence class representative, and update all references to *temp* to refer to *top* instead. The initial equality constraints shown at the top of Figure 2 are clearly not sufficient for inferring correct alias types. For example, the argument *o* of `push` and the return value of `pop` should have the same alias type, yet just looking at the `Stack` class is insufficient to discover this information. Only by reasoning about how objects are stored within the `Link` class can we infer the correct alias types for `Stack`. In our system, this reasoning is done with component and instantiation constraints.

Component Constraints. A component constraint ($o \quad _m v$), read "*v* is a component of *o* with index *m*," means that the type variable *v* represents member *m* of object *o*. Component constraints allow us to keep track of the relationship between a particular stack and the objects and links within that stack, for example. For each member `m` of a class `C`, we generate a component constraint $C \quad _m m$. We generalize the notion of member to any type variable

within a class, so that component constraints are also generated for method arguments, results, and local variables. Figure 2(b) shows the component constraints generated from Figure 1.

Instantiation Constraints. If `C` is a class, an instantiation constraint ($C \leq_v o$), read "*o* is an instance of *C* with index *v*," means that type variable *o* represents an object that is an instance of `C` that is stored in the local variable or field *v*. Instantiation constraints allow us to treat different instances of a class separately; we group instances by the local variable or field that the instance is stored in. Each instance will have its own copy of its local variables and fields in our representation—these are generated by the propagation rules discussed below. For example, different instances of `Stack` can have different actual alias parameters, so that different stacks can hold objects with different owners. For each class member `m` that has declared type `C`, we generate an instantiation constraint $C \leq_m m$.

Instantiation constraints are also used to reason about the relationship between type variables in two different classes. For example, the argument `o` of `push` is assigned to the `_obj` argument of the constructor of the link represented by the type variable *top*. We encode this relationship with the instantiation constraint $_obj \leq_{top} o$, indicating that *o* is the instance of *_obj* inside the *top* link. Here, the index on the instantiation constraint shows how the instance is related to its parent. Thus, for each member `m` that flows to or from a member `n` of another class at a method call or field dereference with receiver `r`, we generate an instantiation constraint $n \leq_r m$. Figure 2(b) shows the instantiation constraints generated from Figure 1.

Component and Instance Uniqueness. In the example program, values flow from the argument `o` of `push` to the `obj` field of `top`, and from the `obj` field of `top` to the result of `pop`. This is represented by the two instantiation constraints $obj \leq_{top} pop$ and $obj \leq_{top} o$ (here

we assume that *_obj* and *get* have already been unified into *obj*). The index *top* common to both these constraints indicates that *pop* and *o* are the same instance of *obj*. Intuitively, *pop* and *o* should be unified, because program values can flow from `o` into `obj` and then back into `pop`. We formalize this intuition with an instance uniqueness rule:

$$a \leq_b c \wedge a \leq_b d \;\Rightarrow\; c = d$$

This rule ensures that two instances of the same type variable that have the same index will be unified. Once *pop* and *o* are unified into *o*, *i* and *i2* will both be instances of *o* with the same index *st*, and so they will be unified as well. An analogous rule is used to ensure that two components of the same type variable with the same index are also unified:

$$a \quad {}_b\, c \wedge a \quad {}_b\, d \;\Rightarrow\; c = d$$

Figure 2(c) shows the example system after solving the initial equality constraints and applying the uniqueness rules.

Constraint Propagation. If `top` is an instance of `Link`, as shown in Figure 2(c), then it ought to have `next` and `obj` components. Furthermore, these components ought to be fresh, distinct from the `next` and `obj` components of any other `Link`. This motivates the component propagation rule:

$$a \quad {}_b\, c \wedge a \leq_I d \;\Rightarrow\; \exists\, e \,.\, d \quad {}_b\, e$$

Applied to *top*, this rule states that since *Link* has a component *next* (*Link* $\quad {}_{next}$ *next*) and *top* is an instance of *Link* (*Link* $\leq_{top}$ *top*), then there must exist some variable *top_next* such that *top_next* is a component of *top* at index *next* (*top* $\quad {}_{next}$ *top_next*). Intuitively, this new variable represents the particular "`next`" link in the `top` field of `Stack`, potentially distinct from the `next` link of any other `Link`.

Now, anything we infer about *next* (for example, if we discover it is equal to some other type variable) must also apply to *top_next*, since *top_next* is just a specialization of *next* that is a component of the *top* instance of *Link*. We encode this intuition with the constraint that *top_next* is an instance of *next*. Then *top_next* will be a transitive instance of *Link*, ensuring that it will gain its own *next* and *obj* components. These constraints are generated with the instance propagation rule:

$$a \quad {}_b\, c \wedge a \leq_I d \wedge d \quad {}_b\, e \;\Rightarrow\; c \leq_I e$$

The precondition for this rule is the conjunction of the precondition and the conclusion of the component propagation rule. Thus, this rule applies whenever a new component constraint is generated. In the case of *top_next*, the rule's conclusion simply states that $next \leq_{top} top_next$.

Avoiding Infinite Propagation. The discussion above suggests that constraint propagation as presented above may never terminate. For example, *top* is a *Link*, so it must have a *next* component *top_next*. But, *top_next* is transitively a *Link* also, so with a couple of instantiation constraint propagations we discover that we need to create *top_next_next*, a *next* component of *top_next*. There must be a way to stop this expansion if the algorithm is to terminate.

We apply the *extended occurs check* to avoid infinite constraint propagation. The extended occurs check rule can be stated as follows:

If $\exists\, L \leq_{i1} a_1 \leq_{i2} \ldots \leq_{iN} R$ and $\exists\, L \quad {}_{c1}\, b_1 \quad {}_{c2} \ldots \quad {}_{cM}\, R$

then $L = R$

Intuitively, this rule states that if one type variable R is both a transitive instance and a transitive component of another type variable L, then we should unify L and R to avoid infinite constraint propagation. In the example, the extended occurs check would discover

that $Link \quad _{next}\ next \wedge Link \leq_{next} next$. Thus, our implementation generates the equality constraint $next$=$Link$, which eliminates the source of the loop.

Figure 2(d) shows the final results of the constraint-based algorithm. As described above, *next* has been unified into *Link*. Also, component propagation has resulted in two components each for *top* and *st*. Due to application of the component and instance uniqueness rules, the components of *top* are itself (just as *Link* is its own component) and *o*, while the components of *st* are *i* and a new node, *st_top*. Like *top*, of which it is an instance, *st_top* has two components, itself and *i*.

The example constraint system has now reached fixpoint with respect to the constraint propagation and uniqueness rules. *Link* has two components, one of which refers to another *Link* instance; these represent the alias parameters used in Figure 1. *Stack* also has two components; one of these will turn into `Stack`'s alias parameter, and the other will turn into an `owned` annotation, as discussed below. Finally, *StackClient*'s two components will eventually turn into `owned` and `unique` annotations.

Integration With Other Alias Annotations. The algorithm described above can infer alias parameters for each class in the system. However, some of the type variables in the example should actually be given a non-parameter alias type. For example, `temp` and `i2` could be annotated `lent`, and `st` and `i` could be annotated `unique`.

We integrate alias parameter inference with inference of other alias annotations by storing a boolean flag in each node for each possible non-parameter annotation: `lent`, `unique`, `owned`, and `shared`. Below, we discuss how each flag is initialized and propagated as type inference proceeds, and how a final alias annotation is computed from the flags at the end.

The owned flag is initialized to true for each variable that is non-public and is never accessed on a receiver other than this. These constraints are the two base-case semantic requirements for owned methods and fields. When two nodes are merged, the resulting node is owned only if both of the merged nodes were owned.

The shared flag is initialized to true for each static field and each argument and result of a static or native method, as these are the base cases for shared annotations. Whenever a shared node is merged with an unshared one, the resulting node is shared. Furthermore, whenever a component constraint is introduced, if the parent node is shared, then the component node must be marked shared as well—otherwise, there would be no way to express its alias annotation in the final system.

The lent and unique flags are initialized with the result of lent and unique inference, as described above. Lent and unique flags are not modified or propagated during constraint solution.

Final Alias Annotations. The final alias annotations are assigned from the constraint graph so as to make the annotations as precise and flexible as possible. Since lent is the most general annotation, all declarations whose node has a lent flag equal to true are given a lent annotation. Unique is the most precise possible annotation for the remaining declarations, so every remaining declaration whose node has a true unique flag is annotated unique. In order to be sound, we must next make every unmarked declaration whose equivalence class representative (ECR) node has a true shared flag shared. Next, we mark the remaining declarations as owned based on their ECR nodes' owned flags. All remaining declarations must be marked with an alias parameter of the enclosing class; for each class, the different ECR nodes that are components of that class are given letter names a, b, c, and so forth.

In the stack example, the nodes *i2* and *temp* have true lent flags, and so these variables are marked `lent` (note that this is a more optimistic annotation than the one given in Figure 3). The variable `i` is marked `unique` on a basis of node *i*'s flags. In class `Stack`, the ECR node for `top` has a true owned flag, while the ECR node *o* representing members `pop` and `o` is not owned. Thus, top is annotated `owned`, while `pop` and `o` are annotated with a fresh alias parameter `a`. Likewise, `member` and `next` are given fresh alias parameters `a` and `b` in class `Link`.

Declarations that have a class type which is parameterized must be given actual alias parameters that correspond to the formal alias parameters of the class. Because of the way the constraints were set up, the declaration's node will have a component node that is an instance of each formal parameter of the class, and the corresponding actual parameter can be computed from this node: either `owned`, `shared`, or a formal alias parameter of the enclosing class. For example, in class `Stack`, we need to assign actual alias parameters to `top`, `temp`, and the `new` expression. These all share the same ECR node, *top*. But node *top* has two component nodes: itself and *o*. Node *o* corresponds to parameter `a` of `Stack`, and *o* is an instance of *obj* (which is parameter `a` of `Link`), so the `a` is used as an actual of `top` corresponding to the formal parameter `a` of `Link`. Node *top* is `owned`, and is an instance of *Link* (which is parameter `b` of `Link`), so `owned` is used as an actual of `top` corresponding to the formal parameter `b` of `Link`. Thus the inferred type of `top` is `owned Link<a,owned>`, and similar types are inferred for `temp` and the `new` expression.

b. Modifications for Interactive Inference

The modifications I have made are as follows:

I. Checking user annotations

The tool stores any user annotations that are in the source code, including formal parameters of classes and interfaces and alias types and actual parameters of variables.

II. Changes to lent and unique inference

Variables that are user-annotated as something other than `lent` are marked as base-case non-`lent` variables during `lent` inference. If the flow constraints indicate that a variable with a user `lent` annotation should be non-`lent`, it is not marked as non-`lent`; however, the tool maintains a mapping between variables with inconsistent user `lent` annotations and the sets of non-`lent` variables that they directly flow to. A representation of the flow graph is also retained in order to enable inference tracing.

The changes made to `unique` inference are similar, except that a variable's `unique` annotation is inconsistent if a non-`unique` variable flows to it rather than from it.

The original inference tool saved time by not propagating non-`unique`ness from lent variables. This worked because `lent` variables could only flow to `lent` variables. However, now that user-annotated `lent` variables can flow to non-`lent` variables, this optimization has been removed.

Finally, code has been added to enable inference tracing as described above.

III. Changes to ownership inference

Any two nodes in the same class with same user ownership annotation are unified immediately, as are any two `shared` nodes at all.

Two new kinds of constraints have been introduced: inequality constraints (indicating that two nodes definitely do not represent the same aliasing behavior) and possible equality constraints (indicating that we may wish to unify two nodes but it is not necessary to do so).

Inequality constraints are added between two nodes in the same class with different user `owned` or parameter annotations, between two nodes in different classes with user `owned` annotations, and between the (now singleton) `shared` node and any node with a user `owned` or parameter annotation.

Originally, if a variable flowed to another variable in the same class, their nodes were always unified. This is no longer done in the case of flow to a lent variable or from a unique variable, since it is legitimate for two objects with different owners to both flow to the same `lent` location; however, it is still necessary for the actual ownership parameters of a variable to be consistent with the parameters of any variable to which it flows. In terms of AliasJava syntax, this means that their corresponding components must be equal. To implement this, we maintain a list of "component-equality classes". Whenever there is flow to a lent variable or from a unique variable, the component-equality classes of the two nodes are merged (every node is initially considered to be in a component-equality class by itself). If at any time two nodes in the same component-equality class have different nodes as components at the same index, an equality constraint is immediately generated between the components.

Lists are maintained of nodes that must or must not be `owned` or `shared` (whether this is due to code or user annotations); inequality constraints are generated between forced `owned`

and forced non-`owned` nodes in the same class, and between the `shared` node and all forced non-`shared` nodes. Since components and instances of `shared` nodes are made `shared`, any node that has a forced non-`shared` component or instance is also put in the list of forced non-`shared` nodes (almost; see immediately below).

Previously, all components of `shared` nodes were made `shared`. This is no longer done when all variables represented by the component are `lent` and `unique`, since a non-`shared` object may flow to a `lent` component or from a `unique` component of a `shared` node; instead, the component is added to a list of "potential `shared`" nodes, meaning that if it ever is unified with a node that is neither `lent` nor `unique`, then it will become `shared`. If an entirely `lent` and `unique` node has a forced non-`shared` component or instance, it is put in a list of "potential non-`shared`" nodes, indicating that if the node is unified with a non-`lent` and non-`unique` node, then it will be moved to the list of forced non-`shared` nodes. Inequality constraints are generated between potential `shared` nodes and forced non-`shared` nodes, and vice versa.

The extended occurs check now generates possible equality constraints instead of equality constraints, in order to make sure that it does not create any unnecessary inconsistencies.

Inequality propagation rules have been introduced to make sure that the graph is never modified so that two inequal nodes must be unified; these rules logically follow from the rules that dictate when two nodes must be unified. The instance uniqueness rule states that a node has no more than one instance at any specific index; from it we obtain the following two inequality propagation rules:

$$A \leq_B C \wedge A \leq_D E \wedge C \neq E \Rightarrow B \neq D$$

$$A \leq_B C \wedge D \leq_B E \wedge C \neq E \Rightarrow A \neq D$$

Likewise, from the component uniqueness rule, we obtain the following two rules:

$$A \rhd_B C \wedge A \rhd_D E \wedge C \neq E \Rightarrow B \neq D$$

$$A \rhd_B C \wedge D \rhd_B E \wedge C \neq E \Rightarrow A \neq D$$

We obtain the following rules from component-equality, where "$x =_C y$" means "x and y are in the same component-equality class":

$$A \rhd_B C \wedge D \rhd_E F \wedge A =_C D \wedge C \neq F \Rightarrow B \neq E$$

$$A \rhd_B C \wedge D \rhd_B E \wedge C \neq E \wedge A =_C F \Rightarrow D \neq F$$

From the constraint propagation rules we obtain the following inequality propagation rule:

$$A \leq_I B \wedge B \rhd_J C \wedge D \rhd_J E \wedge E \leq_I F \wedge C \neq F \Rightarrow A \neq D$$

Other inequality propagation rules could be derived from the constraint propagation rules, but it turns out that only this one is necessary; for example, the following possible rule generates no inequalities that would not be generated by the rules derived from instance uniqueness:

$$A \leq_I B \wedge B \rhd_J C \wedge A \rhd_J D \wedge E \leq_I F \wedge C \neq F \Rightarrow D \neq E$$

There is also an inequality propagation rule based on indirect proof:

$$(A = B \Rightarrow A \neq B) \Rightarrow A \neq B$$

To implement this, if we are considering unifying nodes A and B, then for each C such that $A \neq C$ we propagate $B \neq C$ and for each D such that $B \neq D$ we propagate $A \neq D$ using the other rules; if $A \neq B$ was generated in the process then the unification cannot happen. In such a case, we retain the constraint $A \neq B$ but discard all other generated inequality constraints.

Originally, constraint propagation was not done until after the extended occurs check produces no more results; doing constraint propagation first would defeat the purpose of the

extended occurs check. In the interactive tool, it is necessary to determine when constraint propagation will unify two nodes in order to make sure not to report a conflict when there is none. There are two processes for doing this.

First, we apply constraint propagation as soon as we can when it would not generate a new node; in other words, if $A \rhd_B C$ and $A \leq_D E$, then if $C \leq_D F$ and $E \rhd_B G$ we add the constraint $F = G$; if only one of $C \leq_D F$ and $E \rhd_B G$ is already present we add $E \rhd_B F$ or $C \leq_D G$ respectively. This does not cause infinite looping because it does not generate any new nodes.

Second, we have found that it is sometimes necessary to predict unifications that will be caused by constraint propagation more remotely. For example, consider the following constraints graph:

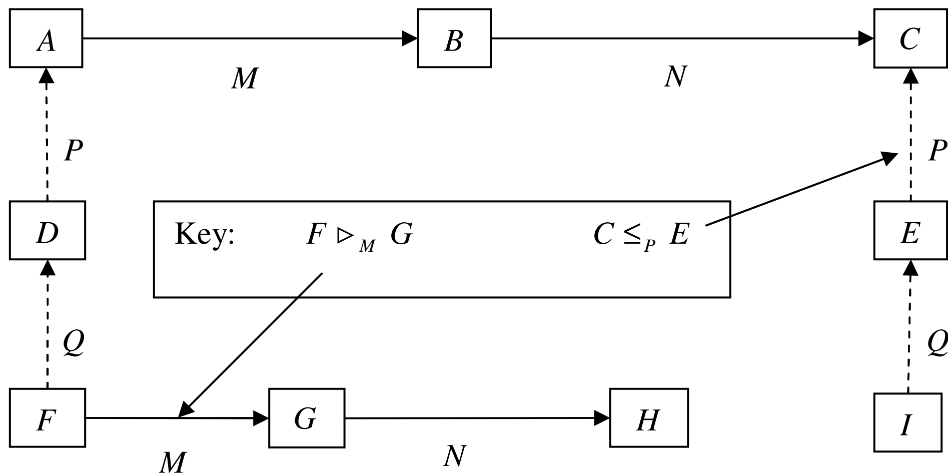

Figure 3: Example of “remote” constraint-propagation-based unification.

Constraint propagation will generate a node J such that $D \rhd_M J$ and $B \leq_P J$. It will then generate the constraints $J \leq_Q G$ and $J \rhd_N E$. Finally, constraint propagation will generate the constraint $H = I$.

To take care of these harder-to-find unifications, we make use of the following statement, which we have not proven but which we believe we could prove:

> Two nodes X and Y will eventually be forced to be unified if and only if:
>
> there exists a path from X to Y
>
> of forward and backward component and instance constraints
>
> such that when traversing the path,
>
> > if a stack is kept for component indexes
> >
> > and another for instance indexes,
> >
> > and traversing a constraint backwards
> >
> > > (i.e. traveling from its right side to its left side)
> > >
> > > adds the index to the corresponding stack
> >
> > and traversing a constraint forwards
> >
> > > (i.e. traveling from its left side to its right side)
> > >
> > > pops an index off the stack
> > >
> > > and asserts that it is the same as the constraint index
> >
> > then when you get to Y, both stacks are empty
> >
> > > and no assertion failure or empty stack failure has occurred.

When the tool is started, a maximum depth at which to check is stored (the depth of the check being half the maximum length of paths that it can find); this value can be specified by a command line argument, otherwise 4 (the minimum depth of a hard-to-find unification) is used. A depth-first search algorithm is used to find unification-inducing paths within this bound, and an equality constraint is generated between the endpoints of each path found; this algorithm is run immediately before first EOC run (so that we know the user annotations are

self-consistent before we start doing unifications that might introduce unnecessary inconsistencies) and periodically thereafter, but not too often since it can be quite time-consuming on large graphs (the current implementation runs it after every 100 unifications based on possible equality constraints). Since we run this algorithm so infrequently, it is necessary to sometimes undo previous changes that we have made due to possible equality constraints. After each run of the path-finding algorithm, we save a backup copy of the constraints graph. We keep a list of possible equality constraints that have been accepted since the last run of the path-finding algorithm. If an equality/inequality conflict is encountered after a backup has occurred, we attempt to recover as follows. If two nodes have recently been unified due to a possible equality constraint, then we replace the last possible equality constraint on the list by an inequality constraint, reload the backup graph, and replay the constraints on the list. Otherwise, if the possible equality constraint list is not empty, then we reload the backup graph and replay using binary search with the path-finding algorithm to find where in the list the problem is; then we negate that constraint and proceed. If no problem is discovered, we assume the algorithm fixed it inadvertently and proceed; this might happen because an equality constraint added by a run of the path-finding algorithm changed the graph in such a way that the inference tool is now able to determine that some pair of nodes should not be unified although this previously could not be determined without running the path-finding algorithm. If an equality/inequality conflict occurs when the list is empty, then we have reached a point where there is a problem we cannot fix: the last run of the path-finding algorithm reported no problems with the graph, and all changes in the graph since then have been necessary, so we cannot attempt to fix the problem by undoing something. In such a situation, the inference tool terminates and reports that it was unable to fix a problem.

The following features have been added to make sure that classes with complete formal parameter lists do not end up with more parameters than they should.

If a possible-equality constraint between two nodes is rejected, then possible-equality constraints are added between corresponding components of the nodes, except where inequality constraints are already present. This exception is necessary to prevent infinite looping; for example, if $X \triangleright_I X$ and $Y \triangleright_I Y$ and $X \neq Y$, and we reject a possible-equality constraint between X and Y, then adding another possible-equality constraint between X and Y would cause the same thing to happen again, and the inference tool would never terminate.

After inference finishes running, if any class's parameter list needs to be reduced, we apply a list of heuristics to that class in order until one produces at least one possible unification; we then add possible-equality constraints to the list of constraints to be dealt with and resume running the inference algorithm. The implemented list of heuristics is:

i. Attempt to unify component-equal nodes.

ii. Attempt to unify return values and corresponding arguments of overloaded methods.

iii. For two variables of the same type, or a type and a variable with that type:

-suggest unifying the two nodes if they are not inequal

-if they are inequal, then suggest unifying their corresponding components

iv. If all else fails, unify any of the class's transitive components that can be unified.

5. Future Research Possibilities

We have not tested the interactive inference tool on any large code bases. While we are fairly confident that the tool works, it would be useful to apply it to at least one large system to determine how useful it is in practice.

If enough of the public interface of a class is annotated by a programmer, it should be possible under certain circumstances to infer ownership types for classes that depend on this class without simultaneously inferring the remaining ownership types for this class. We believe that modifying the inference tool to run on as small a code base at a time as possible would noticeably improve the running time.

As mentioned above, while developing the interactive inference tool we noticed that certain references (such as the `mutex` in `SynchronizedCollection`) are not used in ways that could cause aliasing bugs and could therefore be considered harmless aliases. It may be possible to extend AliasJava to incorporate an annotation for such references.

We were mostly unsuccessful in trying to provide feedback to the user regarding inference of `shared`, `owned`, and parameter annotations. A system for providing useful feedback about this part of the inference process could greatly improve the usefulness of the tool.

The concept of ownership domains [3] is similar to the concept of ownership types but more powerful. At the moment no inference algorithm for ownership domains has been created; thus, while ownership domains are more expressive than ownership types, they may be less practical to apply to large preexisting systems. Inference for ownership domains would make this more powerful system practical enough to be appealing.

6. Related Work

The inference tool described in this paper is based on the original AliasJava inference tool; AliasJava and the original, non-interactive inference algorithm are described in [1] and in more detail in [2].

[4] describes an "algorithm for inferring certain data sharing relationships in Java programs" which "identifies references as *owning*, *borrowing*, or *sharing*"; these correspond roughly to AliasJava's `unique`, `lent`, and `shared` annotations respectively. However, the goal of their inference algorithm is to aid reasoning about when an object may be deallocated or garbage collected, whereas AliasJava's goal is to aid reasoning about where a reference may be aliased.

7. Conclusion

This paper described a new inference algorithm for ownership types which is able to accommodate the presence of some user-specified types and which is able to provide some feedback to the user regarding why it inferred certain types. Using this tool should be less tedious than annotating a program purely by hand, while at the same time producing more usable results than were produced by the original inference tool. This should allow developers to obtain the benefits of ownership types in practice, which would not always be feasible otherwise.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. University of Washington technical report UW-CSE-02-11-01, November 2002.

[3] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. Proc. European Conference on Object-Oriented Programming, June 2004.

[4] Samuel E. Moelius III and Amie L. Souter. An Object Ownership Inference Algorithm and its Application. Drexel University. Available at http://sciris.shu.edu/masplas2004/MASPLAS%20Papers/Paper%206.pdf