Alias Annotations for Program Understanding

Jonathan Aldrich Valentin Kostadinov Craig Chambers

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
+1 206 616-1846

{jonal, valmk, chambers}@cs.washington.edu

Abstract

One of the primary challenges in building and evolving large object-oriented systems is understanding aliasing between objects. Unexpected aliasing can lead to broken invariants, mistaken assumptions, security holes, and surprising side effects, all of which may lead to software defects and complicate software evolution.

This paper presents AliasJava, a capability-based alias annotation system for Java that makes alias patterns explicit in the source code, enabling developers to reason more effectively about the interactions in a complex system. We describe our implementation, prove the soundness of the annotation system, and give an algorithm for automatically inferring alias annotations. Our experience suggests that the annotation system is practical, that annotation inference is efficient and yields appropriate annotations, and that the annotations can express important invariants of data structures and of software architectures.

1. Introduction

Understanding and evolving large software systems is one of the most pressing challenges confronting software engineers today. When evolving a complex system in the face of changing requirements, developers need to understand how the system is organized in order to work effectively. For example, to avoid introducing program defects, programmers need to be able to predict the effect of making a software change. Also, while fixing defects, programmers need to be able to track value flow within a program in order to understand how an erroneous value was produced. In an object-oriented program, all of these tasks require understanding the data sharing relationships within the program. These relationships may be very complex—at worst, a reference could point to any object of compatible type—and current languages do not provide much help in understanding them [HLW+92].

Data sharing problems can also compromise the security of a system. For example, in version 1.1 of the Java standard library, the security system function `Class.getSigners()` returned a pointer to an internal array, rather than a copy. Clients could then modify the array, compromising the security of the "sandbox" that isolates Java applets and potentially allowing malicious applets to pose as trusted code. Existing languages provide poor support for preventing security problems that arise from improper data sharing.

In this paper, we describe and evaluate AliasJava, a type annotation system for specifying data sharing relationships in Java programs. The annotations provide automatically checked documentation about data sharing within a program, while allowing software engineers to program in much the same style as before. We have also applied AliasJava to specify the data sharing relationships within a software architecture, as expressed in the architecture description language ArchJava [ACN02a].

AliasJava's annotations capture several common forms of sharing in object-oriented systems. First, objects are often shared in a structurally bounded way: an object might be shared within the implementation of a subsystem, but not beyond it. In AliasJava, objects that are part of a subsystem's representation are specified with an *owned* type annotation; the subsystem can grant trusted external objects the capability to access its owned state using a simple form of ownership parameterization. Second, objects are sometimes shared in a time-bounded way: an object may be passed as a parameter to a method, which uses the object for the duration of the call, but does not store a persistent reference to the object. AliasJava specifies this kind of time-bounded access capability with a *lent* type annotation. Finally, our type system also includes the best-case *unique* annotation for unshared objects and the worst-case *shared* annotation for objects that have no owning subsystem.

The contributions of this paper are the following:

- a capability-based type annotation system that combines uniqueness and ownership-style encapsulation;
- an implementation in Java and a discussion of issues including concurrency, inner classes, iterators, and casts;
- a formalization of our type annotation system for a subset of Java and a proof outline of several key invariants;
- a novel algorithm for inferring alias annotations; and
- an empirical evaluation of AliasJava on a non-trivial program and on part of the Java collection class library.

The rest of this paper is organized as follows. The next section introduces AliasJava with a series of examples. Section 3 formalizes our type system and outlines proofs of key properties. Section 4 describes our annotation inference algorithm. We evaluate our system in section 5 on a realistic program and on the Java collection libraries. Section 6 discusses related work, and we conclude in section 7.

2. AliasJava

Our type annotation system is motivated by the desire to understand the data sharing patterns in very large software systems. AliasJava annotates all reference types, describing the extent to which that reference is shared. The annotations bound aliasing on the heap structurally: *unique* describes an unshared reference, *owned* objects are assigned an *owner* that controls who

```
class LinkedList {
  private unique Object item;
  private unique LinkedList next;

  public LinkedList(unique Object o,
            unique LinkedList n) {
    item = o; next = n;
  }
  public unique Object getItem() {
    unique Object temp = item;
    item = null;
    return temp;
  }
  public unique LinkedList getNext() {
    unique LinkedList tempNext = next;
    next = null;
    return tempNext;
  }
}

unique LinkedList list =
     new LinkedList(new Object(), null);
list=new LinkedList(new Object(), list);
unique Object o = list.getItem();
list = list.getNext();
```

Figure 1. A linked list class with `unique` links and items

may access that object, and *shared* indicates the worst case of a globally-aliased reference. We also provide a *lent* annotation expressing sharing that is temporally bounded by the length of a method call.

In this section, we present our annotations as a type system for Java programs that provides global guarantees about aliasing. However, adding alias annotations to a large legacy program may require significant effort. Our annotations can also be applied to verify local properties within a subsystem, treating the annotations at the edge of the subsystem as unchecked assertions. We use this methodology in our case studies in Section 5. A promising alternative is inferring alias annotations for a closed subset of the program automatically. Section 4 presents an annotation inference algorithm, and we present early results from a prototype implementation.

Subsection 2.1 describes the AliasJava language through a series of examples. A more precise description of the core annotation system is provided by the formal semantics in section 3. The following two subsections describe the properties guaranteed by AliasJava, and how AliasJava's design works with the features of the full Java language. Subsection 2.4 shows more examples of the language in order to illustrate its expressiveness. We discuss some of the reasoning benefits provided by our annotation system in subsection 2.5.

2.1. Annotations for Data Sharing

Unique. When an object is first created, it 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. Figure 1 illustrates uniqueness through a linked list class where all of the elements and all of the links are **`unique`**.

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. A standard intraprocedural live variable analysis is used to verify this criterion for **`unique`** local variables. When a **`unique`** field is read by a method, that method must set the field to another value before executing any statement (such as a method call or exception-throwing expression) that could result in reading the original value of the field a second time. For example, in Figure 1, the `getItem` method sets the `item` field to **`null`** so that no aliases are created to the **`unique`** value when the item is returned.

```
class Point {
  int x; int y;
  Point(int x, int y) { this.x = x; this.y = y; }
}

class Rectangle {
  private owned Point upperLeft;
  private owned Point lowerRight;

  public Rectangle(unique Point ul,
                   unique Point lr) {
    // ensure Rectangle has non-negative area
    if (ul.x > lr.x || ul.y > lr.y)
      throw new IllegalArgumentException();
    upperLeft = ul;
    lowerRight = lr;
  }
  public unique Point getUpperLeft() {
    return new Point(upperLeft.x, upperLeft.y);
  }
}
```

Figure 2. A `Point` class and a `Rectangle` class that stores its size as a pair of points.

In AliasJava, **`unique`** can be considered a universal source: **`unique`** values can be assigned to a location with any other data sharing annotation. The converse is not true, as the other data sharing annotations do not guarantee that a value is unique. In our capability model, unique is the strongest capability, since unique objects can be assigned to a variable with any other alias annotation.

Owned. Figure 2 shows two classes modeling points and rectangles. The rectangle class represents its shape using two points, one for the upper-left corner of the rectangle and one for the lower-right corner.

A class like `Rectangle` may need to maintain invariants over its state; for example, the code in Figure 2 ensures that the rectangle does not have a negative size, i.e. the upper left-hand point is not below or to the right of the other point.

Maintaining these invariants depends on the lack of external aliases to the `Point` objects that are part of the rectangle's representation. It is not sufficient to make the `Point` fields **`private`**, because aliases to the internal representation could still be exposed. For example, a naïve implementation of the `getUpperLeft` method could expose `Rectangle`'s representation by returning the internal `Point` object rather than a copy. The invariants of `Rectangle` could also be violated if two rectangles accidentally shared the same `Point` objects.

Our **`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. This allows the implementer of `Rectangle` to rely on the fact that external

```
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.member();
  }
  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 member() {
    return obj;
  }
  public link Link<element, link> next() {
    return nxt;
  }
}
```

Figure 3. 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.

objects can only change or see changes to its representation through the rectangle's interface. Owned references may only flow to **`owned`** variables within the scope of the owning object. If, for example, the `getUpperLeft` method returned an alias to the internal point, the compiler would flag the error as a violation of encapsulation.

Ownership parameters. In our capability model, **`owned`** represents a capability that every object has to access its own representation. However, 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 *alias parameter* to the container class, granting that class the capability to reference the element data that are owned by another object. Our system also includes ownership parameterization for methods; an example is shown in section 5.1.

For example, Figure 3 shows a `StackClient` class that 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.

```
class Singleton {
  private static shared Singleton val
       = new Singleton();

  public static shared Singleton get() {
    return val;
  }
  public void doSomething() {
    // application specific code
  }
}

shared Singleton s = Singleton.get();
s.doSomething();
```

Figure 4. A `shared` Singleton object

The stack uses a linked list to store its elements. References to the links in the list should be confined to 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.

Shared. Figure 4 illustrates the Singleton design pattern [GHJ+94], used to create a single instance of an object that is used throughout an application. Singleton objects are intended 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. Figure 5 shows a method that could be part of the `LinkedList` class from Figure 1. This method checks if an integer is stored in a linked list that is made up of **`unique`** `LinkedList` and `Integer` objects. This would be difficult to express with the annotations presented so far, because `contains` would have to destroy the linked list while traversing it in order to avoid creating aliases to the links and elements in the list. Instead, the method uses the **`lent`** annotation to create temporary aliases to the unique objects in the list. These aliases must be destroyed when the `contains` method returns, so that the uniqueness of the linked list is preserved across calls to `contains`.

As shown in this example, **`unique`** objects can be passed as **`lent`** parameters to methods; the called method can pass on the object as a **`lent`** parameter to other methods, but cannot return it or store it in any field. Thus, the **`lent`** annotation preserves all the reasoning about the unique object, but adds a large measure 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,

```
boolean contains(lent LinkedList head, int i) {
  for (lent LinkedList list = head; list != null;
       list = list.next) {
    lent Integer item = (Integer) list.item;
    if (item.intValue() == i)
      return true;
  }
  return false;
}
```

Figure 5. A method that uses a `lent` reference to traverse a linked list looking for an integer

`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. Lent can be thought of a restricted capability that can be used to access an object, but cannot be used to store the object in a field. Lent is the default annotation for method arguments and local variables, and may be omitted.

Other annotations. In designing our annotation system, we chose to focus on precisely specifying the aliasing relationships between objects in the system. Using this criterion, we decided not to include a few annotations that are used in some of the related work. Although package-based confinement [BV99] provides a middle ground between our *shared* and *owned* annotations, we chose not to include it because object ownership is a stronger property and we wanted to keep the system simple. Read-only annotations [NVP98,MP99,BNR01,BR01] can also express useful invariants about a system, but they are not aliasing properties and so were not included in our design. These annotations could probably be added to our system in a natural and orthogonal way.

Summary. Table 1 shows the constraints that our type annotations place on value flow. The various annotations are listed along the left side and the top of the table. An X indicates that data can flow from a location with the annotation on the left to a location with an annotation above. The table shows clearly that `unique` is a universal source (any variable can be assigned a `unique` value), and that `lent` is a universal sink (`lent` variables can be assigned a value with any type annotation). The other type annotations must be kept separate from each other.

2.2. Properties

AliasJava ensures uniqueness and ownership invariants that restrict the aliasing patterns that can occur during program execution. Section 3 proves these invariants for a subset of AliasJava. Our 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`.

Our 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'`.

Table 1. Value flow between alias annotations

		To				
		unique	owned	α	shared	lent
From	unique	X	X	X	X	X
	owned		X			X
	α			X		
	shared				X	X
	lent					X

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. An object delegates a capability to access its `owned` representation by creating a new object and passing `owned` as one of the new object's alias parameters, or by calling a method and passing `owned` as an alias parameter. Because capabilities can only be transferred using the static type parameterization mechanism, AliasJava supports static, source-level human and automated reasoning about which references might alias an `owned` object.

2.3. Java Integration

The Java language has several features that present challenges for an alias control system. We discuss how AliasJava handles of a number of these features below.

Subtyping. We extend Java's declared subtyping relation with our type annotations. When a class is defined, it must provide values for the alias parameters of the classes and interfaces it extends and implements; these values can be any of the alias parameters of the subclass. For example, a class declaration might look like: `class C<α,β,γ> extends B<α,β> implements I<γ>`. When a method or field is overridden, the overriding member must declare its parameters and return value with annotations that exactly match the overridden member, under the alias parameter mapping induced by the inheritance declarations.

This. Since the current object `this` is an implicit argument to all instance methods, its type annotation must be specified. This is done with an annotation that comes immediately after the argument list. This type may be one of `shared`, `unique`, `lent`, or an ownership parameter. Use of `this` within the method must be consistent with its annotation, and at method calls, the receiver is treated as another parameter that must follow the rules for the `this` alias annotation. Because the vast majority of methods and constructors have a `lent` annotation for `this`, `lent` is the default in our system and need not be explicitly specified.

Constructors. Like methods, constructors must specify an alias annotation for `this`. Semantically, we treat a new statement as an allocation of a `unique` object followed by a method call to the constructor for initialization. If the constructor's `this` annotation is `lent`, the allocated object will remain `unique`; if the constructor's `this` annotation is `shared`, the allocated object will be `shared`, etc. Thus, the alias annotation of a newly allocated and constructed object will only be `unique` in the common case where the constructor's `this` annotation is `lent`.

Inner Classes. Inner classes implicitly import the parameters of their surrounding class. The inner class can have its own additional parameters, if necessary. Thus, the qualified type of an inner class is of the form `Package.EnclosingClass<α…>.InnerClass<β…>`. An inner class can refer to the `owned` references of the enclosing class. These values have the type annotation `EnclosingClassName.owned`, while the owned values of the inner class have the annotation `owned`. Anonymous classes defined within a function may not access `unique` or `lent` local variables from the function's scope, because such accesses could create internal persistent references stored in the inner class object, which may violate the type system's invariants.

These special rules do not apply to `static` classes defined within another class. Such classes do not have an implicit pointer to an object of the enclosing class, and so they follow the same rules as ordinary classes, with no special access to their enclosing class.

Static Fields. Static fields are not associated with any particular object instance, and so they cannot be declared with an `owned` or α type annotation (also recall that no field may have a `lent` annotation). Static fields can be `unique` if they are read and written in a way consistent with the `unique` annotation.

Concurrency. Concurrency is largely orthogonal to this work. However, in the presence of concurrency, access to unique fields must be synchronized to prevent two threads from reading a unique variable simultaneously, creating two aliases of a supposedly unique value. We can guarantee uniqueness in the presence of concurrency by ensuring that a `unique` value can flow from an object field into another non-`lent` location only within a block of code synchronized on the object whose field is being dereferenced (or the field's declaring class, in the case of static fields). The field that was read must also be set to another value before the end of the synchronization block.

Casts. Because a class may extend a class that has fewer parameters, alias parameters may be hidden by subsumption. Thus the programmer may have a variable `o` of type `Object`, which has no alias parameters, but may want to cast it down to a `List` type that does have parameters with the expression `(List<owned>)o`. In order to preserve soundness, the runtime system must check both that object `o` is of type `List`, and also that the `List`'s alias parameter is `owned`.

In our implementation, each parameterized class stores the actual owner object for each of its parameters. Note that we do not need to store the owner of each object in the system; our system incurs a small space overhead only for objects that are parameterized. This run-time information is assigned at object-creation time. When a parameter is bound to `owned`, the creating object `this` is recorded to show which object corresponds to the formal parameter. When a class is created with a parameter α, the run-time owner for the corresponding parameter of the creating object is used to discover which object corresponds to the parameter. This run-time parameter information is also passed to methods that have alias parameters.

We cannot add a field to store the ownership parameters for arrays, so we use a global hash table to keep track of the relationship between arrays and their ownership parameters. We use weak references as the keys in the hash table, so that the arrays (and their ownership information) can be reclaimed by the garbage collector when it is otherwise unreachable.

When an object is cast to a parameterized type, the run-time owner for each of its parameters is checked against the corresponding owner specified in the cast, and an `AliasCastException` is thrown if the check does not succeed. In this way, AliasJava supports upcasts and downcasts in a way that does not violate the semantics of the type annotations.

Arrays. An array must be given an alias type for each array dimension. The alias type of the array itself is given by the overall modifier for the array type, while the modifier for each dimension of the array is nested in the corresponding brackets. For example, the variable declaration `unique Stack<β>[owned][α] array` refers to a `unique` array of `owned` arrays of α stacks that hold objects of alias type β. An array dereference of the form `array[0]` would have type `owned Stack<β>[α]`.

Following Java, we support covariant subtyping for arrays. In order to preserve type soundness, we must do a run-time alias check whenever an object is stored into an array, to ensure that the dynamic alias parameters of the object are compatible with the dynamic alias parameters of the array. This check uses the same run time alias annotation information that is used to support sound casts, as discussed above.

The Java Standard Library. We have chosen to implement our system on top of the standard Java Virtual Machine (JVM), and so we did not modify the bytecode of the Java standard library. Unfortunately, this means that Java's reflection interfaces provide a way to get around the alias type system. This could be remedied by replacing the existing reflection library with one that dynamically checks for violations of our alias type system.

Another issue is that since we did not modify the standard library bytecode, our runtime system does not record run-time alias parameter information for parameterized classes and methods created and called by the standard library code. Thus, the parameter information for some methods and objects will be missing at some run-time casts. In our implementation, we always allow these casts to succeed, but a number of other choices are possible in principle.

Implementation. We have added support for AliasJava to the ArchJava compiler, which is publicly available at the ArchJava website [Arc02]. Our implementation is based on the Barat compiler infrastructure [BS98].

2.4. Examples

In this subsection, we present a number of examples that demonstrate the expressiveness of our annotation system.

```
interface Iterator<element> {
  element Object next();
}

public class List<element> {
  private owned Link<element, owned> front;
  void add(element Object e) { ... }
  unique Iterator<element> iterator() {
    return new ListIter<element,owned>(front);
  }
}

class ListIter<element, link>
      implements Iterator<element> {
  private link Link<element, link> cur;
  public element Object next() {
    element Object e = cur.o;
    cur = cur.next;
    return e;
  }
}
```

Figure 6. A `List` class and an iterator over the list

2.4.1. Iterators

Iterators are a challenge to many alias control systems. Figure 6 shows how a `List` class can be defined to return an `Iterator` object that can access its internal representation (the links in the list) without exposing that representation to clients. When the `List` class creates a `ListIter`, it instantiates the second alias parameter of `ListIter` with **`owned`**, thereby delegating a capability to access the list's representation. The `ListIter` is then returned as an object of type `Iterator`, which hides access to the links in the list. Clients of the `Iterator` cannot access these links through the `Iterator` interface, nor can they cast the `Iterator` to `ListIter`, because the `List` has not given them a capability to access its representation.

2.4.2. Uniqueness and Ownership

The combination of the **`unique`** annotation with ownership annotations is crucial to the expressiveness of our annotation system; it allows us to express important idioms that neither class of annotation system could alone. For example, the `Lexer` class in Figure 7 accepts an input stream that becomes part of its representation. The implementation of the `Lexer` relies on the state of the `InputStream`, and therefore the specification of `Lexer` should require that external clients do not modify the state of the stream after passing it to the lexer.

In AliasJava, the `InputStream` argument to `Lexer`'s constructor is **`unique`**, forcing the client to give up its other non-**`lent`** references to the stream. The `InputStream` is then captured into the lexer as an **`owned`** reference, ensuring that persistent aliases to the stream cannot escape the lexer's scope.

2.4.3. Architectural Styles

We have developed ArchJava, an extension to Java that enables developers to express the software architecture of large object-oriented software systems [ACN02a]. The initial version of ArchJava specified only control flow between architectural components; communication though data sharing remained unspecified, reducing the value of the architectural specifications. Our alias annotation system allows us to extend ArchJava architectures to include a specification of data sharing between components. In this subsection, we show how alias annotations can express important invariants of two common *architectural styles* discussed by Garlan and Shaw [GS93].

```
public class Lexer {
  owned InputStream stream;
  Lexer(unique InputStream s) {
    stream = s;
  }
  unique Token getToken() { ... }
}

void lexerClient() {
  unique InputStream stream =
                 new FileInputStream(file);
  unique Lexer l = new Lexer(stream);
  l.getToken();
}
```

Figure 7. A `Lexer` class that uses an `InputStream` as part of its representation. The `InputStream` is passed to the constructor as a `unique` reference.

Pipe and Filter Architectures. Figure 8 shows a pipe and filter architecture, in which the architectural components are filters that accept a stream of data along an input pipe and produce a new stream of data along an output pipe. The example shows two component classes, which are used to define architectural structure in the ArchJava language. The components communicate with each other through *ports*. For example, the `Filter` component below accepts data on its input port, processes the data, and sends the new data out its output port. In addition to ordinary methods, ports may have *requires* methods that represent the interface of a connected component.

In this example, the `Filter` invokes the `accept` method on its output port, which will result in invoking the `accept` method of the filter at the other end of the pipe. The `PipeAndFilter` component class defines an architecture by declaring a set of final fields that hold its subcomponents, and connecting the ports of these components with connections. Connections bind the requires methods in the port of one component to the methods of the same name implemented in the port of another component.

An important invariant of this architectural style is that the filters do not share state; they communicate only through the pipes connecting them. The alias annotations in the system express and enforce this invariant. Because the `Source`, `Filter`, and `Sink` components have no alias parameters, they cannot directly share any data.[1] The **`unique`** annotations in the ports express the invariant that when a data structure is passed from one filter to another, the first filter gives up all references to the data.

This example also shows the practical importance of combining uniqueness and ownership in our annotation system. The data passed between components might not be a simple object, but could be a complex data structure that includes multiple internal objects with nontrivial internal aliasing patterns. A type system with only uniqueness could express passing a unique reference to a data structure between components, but could not express the constraint that aliasing is allowed within the data structure but not beyond it. Similarly, a system with only object ownership could

[1] We are ignoring **`shared`** annotations, but widespread use of these is poor practice and could be flagged by the compiler.

```
component class Filter {
 public port in {
   void accept(unique Data d) {
     // process data and send out
     out.accept(process(d));
   }
 }
 public port out {
   requires void accept(unique Data d);
 }
 private unique Data process(unique Data d) {...}
}

public component class PipeAndFilter {
  private final owned Source source = ...;
  private final owned Filter filter = ...;
  private final owned Sink sink = ...;
  connect source.out, filter.in;
  connect filter.out, sink.in;
}
```

Figure 8. A pipe and filter architecture implemented in ArchJava with alias annotations.

```
public component class Architecture {
  private final owned Blackboard<owned>store=...;
  private final owned Module1<owned> m1 = ...;
  private final owned Module2<owned> m2 = ...;
  connect m1.data, store.data;
  connect m2.data, store.data;
}

public component class Blackboard<data_owner> {
  public port data {
    requires void notify(lent Message change);

    data_owner Data getData(lent Spec spec);
    void update(data_owner Data d);
  }
}
```

Figure 9. A blackboard architecture expressed in ArchJava with alias annotations.

express the limited scope of aliasing within the passed data structure, but could not express the architectural invariant that the first component does not retain any references to the data structure.

Blackboard Architectures. Figure 9 shows a blackboard architectural style, where computational components surround a central data store. The components in a blackboard architecture communicate exclusively by modifying shared state in the data store. Component actions are triggered by changes to the data store made by other components.

In the `Architecture` component class, the connections show the control flow between the computational components and the data store. These control-flow connections specify that components `m1` and `m2` do not call each other's methods directly, but instead communicate only through method calls to the store—and this specification is verified by ArchJava's type system [ACN02b]. The alias annotations, in turn, describe the data sharing relationships between the components. A glance at the `Architecture` code shows that the `store`, `m1`, and `m2` components all share the same alias parameter.

The interface of the data store shows in more detail how data structures are shared between different parts of the architecture. In its `data` port, the data store defines a **`requires`** method that it calls to notify clients whenever data has changed. This method passes a change message to the computational components; this message is **`lent`**, indicating that the clients may not store persistent references to it.

The data store also implements two methods allowing clients to get data and to update the store. Here, the specification of what data is requested is a **`lent`** parameter of `getData`, but the returned data is annotated with the `data_owner` parameter, indicating that it is shared persistently between different components in the architecture.

2.5. Reasoning about Data Sharing

One criterion for evaluating the alias annotation system is, does it help in reasoning about data sharing? In this subsection, we consider the reasoning benefits of our alias annotation system by discussing how the annotations can help programmers answer software maintenance questions that are difficult to answer in existing Java programs.

What parts of the program might be affected by a change to a data structure? This question often comes up when the system must be evolved to meet changing requirements. In general, answering it requires identifying all parts of the program that could refer to the changed data. Confronting this task by tracing through the program manually is tedious and error-prone.

Our alias annotations can give concrete aid in answering this question. If the reference to the modified data is **`unique`**, only the parts of the program to which the unique reference flows can be affected. If the reference to the modified data is **`owned`**, the scope of the change is limited to the current object and its delegates, while a reference annotated with an alias parameter indicates the need to look in the enclosing object to understand sharing patterns. A **`lent`** reference indicates that the current data structure is part of a different object's representation, and it suggests that the caller and callee need to agree on a contract that specifies any intended modifications to the data. References with a **`shared`** annotation are as challenging to reason about as ordinary Java references, but we hope these references will be rare in practice.

What components might this component communicate with? It is important to answer this question when making changes to a large software system. The earlier ArchJava language design makes control flow communication between components explicit in the connections between components. AliasJava's annotations complement ArchJava by making communication through shared data explicit, as shown in Figures 8 and 9.

How difficult would it be to distribute a system across two machines? This question might be important if a system must be scaled beyond the resources of a single machine. Unfortunately, data sharing between components poses challenges for effectively distributing legacy applications. Alias annotations in the system's architecture can help programmers to anticipate the issues likely to come up when distributing a program across multiple machines. For example, if objects annotated **`lent`** or **`unique`** are passed between components that will be distributed across a network, the objects can probably be passed by value between the two distributed components. On the other hand, if the alias

```
CL  ::= class C<α̅,β̅> extends D<α̅> {T̅ f̅; M̅}

M   ::= T m(T̅ x̅) T { return e; }

e   ::= x
    |   new C<α̅>()
    |   e.f
    |   e.f = e, e
    |   (T)e
    |   e.m(e̅)
    |   error
    |   unique(x)
    |   unique(e.f)
    |   A(v)

v   ::= null
    |   ℓ

T   ::= A C<p̅>
    |   NULL
    |   ERROR

A, B    ::= lent | unique | p
p, q    ::= owned | α | ℓ

S   ::= ℓ → C<ℓ̅>(v̅)
Γ   ::= x → T
Σ   ::= ℓ → T

ℓ   ∈ Locations
α,β ∈ Parameters
```

Figure 10. AliasFJ Syntax

annotations in the architecture indicate persistent sharing between components that will be distributed, either a solution using remote object references or extensive refactoring of the source code will be necessary.

3. Formalization

We would like to use formal techniques to prove that the type system is safe, and preserves the intended aliasing invariants. A standard technique, exemplified by Featherweight Java [IPW99], is to formalize a core language that captures the key typing issues while ignoring complicating language details. We have formalized AliasJava as AliasFJ, a core language based on Featherweight Java (FJ).

3.1. AliasFJ

Syntax. Figure 10 presents the syntax of AliasFJ. The metavariables C, D and E range over class names; A ranges over alias annotations; p and q range over actual alias parameters; α and β range over formal alias parameters; T and U range over types; f and g range over fields; v ranges over values; e ranges over expressions; ℓ ranges over locations; S ranges over stores; and M ranges over methods. As a shorthand, we use an overbar to represent a sequence. We assume a fixed class table *CT* mapping classes to their definitions. A program, then, is a pair (*CT*, e) of a class table and an expression.

As in Featherweight Java, AliasFJ omits interfaces, inner classes, and some statement and expression forms. AliasFJ does not have static fields, so we omit the **shared** alias type, which can be considered a special case of parameterization where the owning object is the entire program. These changes make our type soundness proof shorter, but do not materially affect it otherwise.

$$\texttt{C} <: \texttt{C} \qquad \text{(CLASS-REFLEX)}$$

$$\frac{\texttt{C} <: \texttt{D} \quad \texttt{D} <: \texttt{E}}{\texttt{C} <: \texttt{E}} \qquad \text{(CLASS-TRANS)}$$

$$\frac{CT(\texttt{C}) = \textbf{class}\ \texttt{C}<\overline{\alpha}>\ \textbf{extends}\ \texttt{D}<\overline{\beta}>\texttt{...}}{\texttt{C} <: \texttt{D}} \qquad \text{(CLASS-EXTENDS)}$$

$$\frac{\begin{array}{c}\texttt{C} <: \texttt{D} \\ \texttt{A} = \textbf{unique} \vee \texttt{B} = \textbf{lent} \vee \texttt{A} = \texttt{B}\end{array}}{\texttt{A}\ \texttt{C}<\overline{p},\overline{q}> <: \texttt{B}\ \texttt{D}<\overline{p}>} \qquad \text{(SUBTYPE-ALIAS)}$$

$$\textbf{ERROR} <: \texttt{T} \qquad \text{(SUBTYPE-ERROR)}$$

$$\textbf{NULL} <: \texttt{T} \qquad \text{(SUBTYPE-NULL)}$$

Figure 11. Subtyping Rules

AliasFJ extends Featherweight Java in several ways. Classes are parameterized by a list of alias annotations, and extend another class that has a subsequence of its alias parameters. Because we want to reason about aliasing, we add mutable fields and field assignment to FJ. Therefore, a store S maps locations ℓ to their contents: the class of the object and the values stored in its fields. We will write S[ℓ] to denote the store entry for ℓ, and S[ℓ,*i*] to denote the value in the *i*th field of S[ℓ]. Functional store updates are abbreviated $\texttt{S}[\ell \rightarrow \texttt{C}<\overline{\ell}>(\overline{\texttt{v}})]$. The store also holds the actual alias parameters for each location, in order to check run-time casts properly.

Classes define a set of fields $\overline{\texttt{f}}$ and methods $\overline{\texttt{M}}$. Expressions include variables, object creation expressions, field reads and writes, casts, and method calls. We also include an error expression, representing failed casts and null dereferences.

In the compiler for the full language, a live variable analysis identifies the last use of unique variables automatically. AliasFJ models the results of this analysis explicitly by marking a single unique read of a variable with a **unique** tag. Similarly, the compiler for the full language performs an analysis to determine that unique fields are overwritten immediately after being read. Instead of modeling this analysis formally, AliasFJ provides a destructive read operation (again, identified by the **unique** tag) that overwrites the field with **null** after every read.

Values represent irreducible computational results, and include locations in the store and a distinguished **null** location. Different references to the same location in the program may have different alias annotations; for example, there might be some references to a location annotated **lent** and others annotated **unique**. Therefore, values within expressions are tagged with an alias annotation A.

Types. Ordinary types consist of an alias annotation A and a class name parameterized with annotations p. We also include types representing **NULL** and **ERROR**. Annotations may be **lent**, **unique**, **owned**, or a parameter p. Actual alias parameters in the source text must be parameters α of the enclosing class, or **owned**. However, during reduction, these parameters may be replaced with locations ℓ, indicating the object that corresponds to that actual alias parameter. Thus, we include locations in the type syntax so that we can give alias types to expressions in an executing program.

$$\frac{\ell \notin \mathit{domain}(\mathtt{S}) \qquad \mathtt{S}' = \mathtt{S}[\ell \rightarrow \mathtt{C}<\overline{\ell}>(\overline{\mathbf{null}})]}{\mathtt{S} \vdash \mathbf{new}\ \mathtt{C}<\overline{\ell}>() \rightarrow \mathbf{unique}(\ell),\ \mathtt{S}'} \quad (\text{R-NEW})$$

$$\frac{\mathtt{S}[\ell] = \mathtt{C}<\overline{\ell}>(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{T}_i = \mathtt{A}_i\ \mathtt{D}_i<\overline{\beta}> \qquad \mathtt{A}_\mathtt{R} = [\mathbf{lent/unique}]\,[\ell/\mathbf{owned}]\ \mathtt{A}_i}{\mathtt{S} \vdash \mathtt{A}(\ell).\mathtt{f}_i \rightarrow \mathtt{A}_\mathtt{R}(\mathtt{v}_i),\ \mathtt{S}} \quad (\text{R-READ})$$

$$\frac{\mathtt{S}[\ell] = \mathtt{C}<\overline{\ell}>(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{T}_i = \mathbf{unique}\ \mathtt{C}<\overline{\alpha}> \qquad \mathtt{S}' = \mathtt{S}[\ell \rightarrow \mathtt{C}<\overline{\ell}>([\mathbf{null}/\mathtt{v}_i]\overline{\mathtt{v}})]}{\mathtt{S} \vdash \mathbf{unique}(\mathtt{A}(\ell).\mathtt{f}_i) \rightarrow \mathbf{unique}(\mathtt{v}_i),\ \mathtt{S}'} \quad (\text{R-UNIQUEREAD})$$

$$\frac{\mathtt{S}[\ell] = \mathtt{C}<\overline{\ell}>(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathtt{S}' = \mathtt{S}[\ell \rightarrow \mathtt{C}<\overline{\ell}>([\mathtt{v}/\mathtt{v}_i]\overline{\mathtt{v}})]}{\mathtt{S} \vdash \mathtt{A}(\ell).\mathtt{f}_i = \mathtt{v},\ \mathtt{e} \rightarrow \mathtt{e},\ \mathtt{S}'} \quad (\text{R-WRITE})$$

$$\frac{\mathtt{S}[\ell] = \mathtt{D}<\overline{\ell}>(\overline{\mathtt{v}}) \qquad \mathtt{A}\ \mathtt{D}<\overline{\ell}>\ <:\ \mathtt{T}_\mathtt{C}}{\mathtt{S} \vdash (\mathtt{T}_\mathtt{C})\mathtt{A}(\ell) \rightarrow \mathtt{A}(\ell),\ \mathtt{S}} \quad (\text{R-CAST})$$

$$\frac{\begin{array}{c} \mathtt{S}[\ell] = \mathtt{C}<\overline{\ell}>(\overline{\mathtt{v}_\mathtt{f}}) \qquad \mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha}>\ldots \\ \mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e}) \qquad \overline{\mathtt{v}_\mathtt{lent}} = [\mathbf{lent/unique}]\ \overline{\mathtt{v}} \\ \mathtt{this}_\mathtt{lent} = [\mathbf{lent/unique}]\ \mathtt{A}_0(\ell) \\ \mathtt{e}' = [\overline{\mathtt{v}}/\overline{\mathbf{unique}(\mathtt{x})},\ \mathbf{unique}(\ell)/\mathbf{unique}(\mathbf{this}), \\ \overline{\ell}/\overline{\alpha},\ \ell/\mathbf{owned},\ \overline{\mathtt{v}_\mathtt{lent}}/\overline{\mathtt{x}},\ \mathbf{this}_\mathtt{lent}/\mathbf{this}]\ \mathtt{e} \end{array}}{\mathtt{S} \vdash \mathtt{A}_0(\ell).\mathtt{m}(\overline{\mathtt{v}}) \rightarrow \mathtt{e}',\ \mathtt{S}} \quad (\text{R-INVK})$$

Figure 12. AliasFJ Evaluation Rules

Subtyping Rules. AliasFJ's subtyping rules are given in Figure 11. Class subtyping is defined by the reflexive, transitive closure of the immediate subclass relation given by the **extends** clauses in *CT*. We require that there are no cycles in the induced subtype relation. The subtyping relationship between ordinary types follows that of classes. The rule encodes the alias annotation semantics where **unique** is a subtype of any other annotation, **lent** is a supertype of any other annotation, and all other annotations must match exactly. Also, the alias parameters of the supertype must be a subsequence of the subtype's parameters. Finally, any expression can have an **error** or **null** subexpression, and so **ERROR** and **NULL** are subtypes of all other types.

Evaluation Rules. The evaluation relation, defined by the reduction rules in Figure 12, has the form $\mathtt{S} \vdash \mathtt{e} \rightarrow \mathtt{e}',\mathtt{S}'$, read "In the context of store S, expression e reduces to expression e′ in one step, producing the new store S′." We write →* for the reflexive, transitive closure of →. Most of the rules are standard; the interesting features are how they manipulate the alias type system. The R-NEW rule reduces a new expression into a unique reference to a fresh location. The store is extended at that location to refer to a class with the same type and alias parameters, with all **null** fields.

There are two rules for field reads. The R-READ rule applies to normal reads of a field $\mathtt{f}_i$; it looks up the receiver in the store, identifies the *i*th field. The result is the value at field position *i* in the store. The rule derives the annotation for the resulting value from the alias annotation from the type of the *i*th field of the receiver. Because this is not a unique field read, if the field was annotated with **unique** then the resulting value will be annotated with **lent**. We denote this substitution with [**lent/unique**]A, meaning that all occurrences of **unique** in A are replaced with **lent**. Similarly, if the field was annotated with **owned**, the dynamic owner of that field is the actual receiver ℓ, and so we replace any **owned** annotations with ℓ.

$$\mathtt{S} \vdash (\mathtt{T}_\mathtt{C})\mathtt{A}(\mathbf{null}) \rightarrow \mathtt{A}(\mathbf{null}),\ \mathtt{S} \quad (\text{R-CASTNULL})$$

$$\frac{\mathtt{S} \vdash \mathtt{e} \rightarrow \mathtt{e}',\ \mathtt{S}'}{\mathtt{S} \vdash [\mathbf{unique}]\mathtt{e}.\mathtt{f}_i \rightarrow \mathtt{e}'.\mathtt{f}_i,\ \mathtt{S}'} \quad (\text{RC-READ})$$

$$\frac{\mathtt{S} \vdash \mathtt{e}_1 \rightarrow \mathtt{e}_1',\ \mathtt{S}'}{\mathtt{S} \vdash \mathtt{e}_1.\mathtt{f}_i = \mathtt{e}_2,\ \mathtt{e}_3 \rightarrow \mathtt{e}_1'.\mathtt{f}_i = \mathtt{e}_2,\ \mathtt{e}_3,\ \mathtt{S}'} \quad (\text{RC-WRITE1})$$

$$\frac{\mathtt{S} \vdash \mathtt{e}_2 \rightarrow \mathtt{e}_2',\ \mathtt{S}'}{\mathtt{S} \vdash \mathtt{v}_1.\mathtt{f}_i = \mathtt{e}_2,\ \mathtt{e}_3 \rightarrow \mathtt{v}_1.\mathtt{f}_i = \mathtt{e}_2',\ \mathtt{e}_3,\ \mathtt{S}'} \quad (\text{RC-WRITE2})$$

$$\frac{\mathtt{S} \vdash \mathtt{e}_3 \rightarrow \mathtt{e}_3',\ \mathtt{S}'}{\mathtt{S} \vdash \mathtt{v}_1.\mathtt{f}_i = \mathtt{v}_2,\ \mathtt{e}_3 \rightarrow \mathtt{v}_1.\mathtt{f}_i = \mathtt{v}_2,\ \mathtt{e}_3',\ \mathtt{S}'} \quad (\text{RC-WRITE3})$$

$$\frac{\mathtt{S} \vdash \mathtt{e} \rightarrow \mathtt{e}',\ \mathtt{S}'}{\mathtt{S} \vdash (\mathtt{T})\mathtt{e} \rightarrow (\mathtt{T})\mathtt{e}',\ \mathtt{S}'} \quad (\text{RC-CAST})$$

$$\frac{\mathtt{S} \vdash \mathtt{e} \rightarrow \mathtt{e}',\ \mathtt{S}'}{\mathtt{S} \vdash \mathtt{e}.\mathtt{m}(\overline{\mathtt{e}}) \rightarrow \mathtt{e}'.\mathtt{m}(\overline{\mathtt{e}}),\ \mathtt{S}'} \quad (\text{RC-INVK1})$$

$$\frac{\mathtt{S} \vdash \mathtt{e}_i \rightarrow \mathtt{e}_i',\ \mathtt{S}'}{\begin{array}{c}\mathtt{S} \vdash \mathtt{v}.\mathtt{m}(\mathtt{v}_1..\mathtt{v}_{i-1},\mathtt{e}_i,\mathtt{e}_{i+1}..\mathtt{e}_n) \rightarrow \\ \mathtt{v}.\mathtt{m}(\mathtt{v}_1..\mathtt{v}_{i-1},\mathtt{e}_i',\mathtt{e}_{i+1}..\mathtt{e}_n),\ \mathtt{S}'\end{array}} \quad (\text{RC-INVK2})$$

$$\mathtt{S} \vdash [\mathbf{unique}]\mathtt{A}(\mathbf{null}).\mathtt{f}_i \rightarrow \mathbf{error},\ \mathtt{S} \quad (\text{RE-READNULL})$$

$$\mathtt{S} \vdash \mathtt{A}(\mathbf{null}).\mathtt{f}_i = \mathtt{v},\ \mathtt{e} \rightarrow \mathbf{error},\ \mathtt{S}' \quad (\text{RE-WRITENULL})$$

$$\frac{\mathtt{S}[\ell] = \mathtt{D}<\overline{\ell}>(\overline{\mathtt{v}}) \qquad \mathtt{A}\ \mathtt{D}<\overline{\ell}>\ \not<:\ \mathtt{T}_\mathtt{C}}{\mathtt{S} \vdash (\mathtt{T}_\mathtt{C})\mathtt{A}(\ell) \rightarrow \mathbf{error},\ \mathtt{S}} \quad (\text{RE-CASTFAIL})$$

$$\mathtt{S} \vdash \mathtt{A}(\mathbf{null}).\mathtt{m}(\overline{\mathtt{v}}) \rightarrow \mathbf{error},\ \mathtt{S} \quad (\text{RE-INVKNULL})$$

Figure 13. AliasFJ Congruence and Error Rules

The R-UNIQUEREAD rule is similar, but applies to unique reads. Here, the result is always a value with a **unique** annotation, but the value of the field that was read is updated to **null** in the store. This reflects the "destructive read" semantics, which models our user-level language's requirement that **unique** fields be updated after unique reads.

The R-WRITE rule is straightforward, updating the *i*th field of the receiver object with the value written to field $\mathtt{f}_i$. As in Java, the R-CAST rule checks that the cast expression is a subtype of the cast type. Note, however, that in AliasFJ this check also verifies that the alias parameters match, doing an extra run-time check that is not present in Java.

The invocation rule uses the *mbody* helper function (defined in Figure 16) to determine the correct method body to invoke. The method invocation is replaced with the appropriate method body. Several substitutions are made into the body to reflect the method

$$\frac{\mathtt{T} = [\mathbf{lent/unique}]\ \Gamma(\mathtt{x})}{\Gamma,\Sigma \vdash \mathtt{x}:\mathtt{T}} \qquad (\text{T-VAR})$$

$$\Gamma,\Sigma \vdash \mathbf{unique}(\mathtt{x}) : \Gamma(\mathtt{x}) \qquad (\text{T-UVAR})$$

$$\frac{\Sigma(\ell) = \mathtt{C}<\overline{\ell}>}{\Gamma,\Sigma \vdash \mathtt{A}(\ell) : \mathtt{A}\ \mathtt{C}<\overline{\ell}>} \qquad (\text{T-LOC})$$

$$\Gamma,\Sigma \vdash \mathbf{new}\ \mathtt{C}<\overline{p}>() : \mathbf{unique}\ \mathtt{C}<\overline{p}> \qquad (\text{T-NEW})$$

$$\frac{\Gamma,\Sigma \vdash \mathtt{e}:\mathtt{A}\ \mathtt{D}<\overline{q}>}{\Gamma,\Sigma \vdash (\mathtt{A}\ \mathtt{C}<\overline{p}>)\mathtt{e}:\mathtt{A}\ \mathtt{C}<\overline{p}>} \qquad (\text{T-CAST})$$

$$\Gamma,\Sigma \vdash \mathbf{error} : \mathtt{ERROR} \qquad (\text{T-ERROR})$$

$$\Gamma,\Sigma \vdash \mathbf{null} : \mathtt{NULL} \qquad (\text{T-NULL})$$

$$\frac{\begin{array}{c}\Gamma,\Sigma \vdash \mathtt{e}:\mathtt{A}\ \mathtt{C}<\overline{p}> \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ (\mathtt{T}_i = \mathbf{owned}\ \ldots\ \wedge\ \mathtt{e}\text{ a variable}) \Rightarrow \mathtt{e} = \mathbf{this} \\ \mathtt{T}_\mathtt{R} = [\mathbf{lent/unique}]\ \mathit{inst}(\mathtt{T}_i, \mathtt{A}\ \mathtt{C}<\overline{p}>, \mathtt{e}_0)\end{array}}{\Gamma,\Sigma \vdash \mathtt{e}.\mathtt{f}_i : \mathtt{T}_\mathtt{R}} \qquad (\text{T-FIELD})$$

$$\frac{\begin{array}{c}\Gamma,\Sigma \vdash \mathtt{e}:\mathtt{A}\ \mathtt{C}<\overline{p}> \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \mathtt{T}_i = \mathbf{unique}\ \mathtt{D}<\overline{\beta}> \qquad \mathtt{T}_\mathtt{R} = \mathit{inst}(\mathtt{T}_i, \mathtt{A}\ \mathtt{C}<\overline{p}>, \mathtt{e}_0)\end{array}}{\Gamma,\Sigma \vdash \mathbf{unique}(\mathtt{e}.\mathtt{f}_i) : \mathtt{T}_\mathtt{R}} \qquad (\text{T-UFIELD})$$

$$\frac{\begin{array}{c}\Gamma,\Sigma \vdash \mathtt{e}_0:\mathtt{A}\ \mathtt{C}<\overline{p}> \qquad \Gamma,\Sigma \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \Gamma,\Sigma \vdash \mathtt{e}_2 : \mathtt{T}_2 \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}} \\ \mathtt{T}_1 <: \mathit{inst}(\mathtt{U}_i, \mathtt{A}\ \mathtt{C}<\overline{p}>, \mathtt{e}_0) \\ (\mathtt{U}_i = \mathbf{owned}\ \ldots\ \wedge\ \mathtt{e}_0\text{ a variable}) \Rightarrow \mathtt{e}_0 = \mathbf{this}\end{array}}{\Gamma,\Sigma \vdash \mathtt{e}_0.\mathtt{f}_i = \mathtt{e}_1,\ \mathtt{e}_2 : \mathtt{T}_2} \qquad (\text{T-WFIELD})$$

$$\frac{\begin{array}{c}\Gamma,\Sigma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Gamma,\Sigma \vdash \overline{\mathtt{e}}:\overline{\mathtt{U}} \\ \mathtt{T}_0 = \mathtt{A}\ \mathtt{C}<\overline{p}> \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathtt{T}_\mathtt{this} \times \overline{\mathtt{T}} \to \mathtt{T}_\mathtt{R} \\ \overline{\mathtt{U}}\ <:\ \mathit{inst}(\overline{\mathtt{T}}, \mathtt{T}_0, \mathtt{e}_0) \qquad \mathtt{T}_0 <:\ \mathit{inst}(\mathtt{T}_\mathtt{this}, \mathtt{T}_0, \mathtt{e}_0) \\ \mathtt{T} = \mathit{inst}(\mathtt{T}_\mathtt{R}, \mathtt{T}_0, \mathtt{e}_0) \\ (\mathbf{owned} \in \mathit{mtype}(\mathtt{m},\mathtt{C})\ \wedge\ \mathtt{e}_0\text{ a variable}) \Rightarrow \mathtt{e}_0 = \mathbf{this}\end{array}}{\Gamma,\Sigma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{T}} \qquad (\text{T-INVK})$$

Figure 14. AliasFJ Typechecking

argument and receiver values. First of all, any occurrences of formal alias parameters α of the enclosing class are replaced with the actual alias parameters ℓ of the receiver value. Second, the formal parameters of the method `x` as well as the variable `this` are replaced with the actual values passed in. This substitution involves some subtlety, however, because if one of the parameters is annotated **`unique`**, it would not be sound to replace all occurrences of that parameter with the **`unique`** value. Instead, only the unique read of the parameter is replaced with the unchanged argument value; the other non-unique reads are replaced with a modified argument value where **`unique`** annotations have been replaced with **`lent`**.

A set of congruence rules, defined in Figure 13, allows reduction to proceed in the order of evaluation defined by Java. The figure also defines error rules representing casts that fail and null pointer dereferences. The rules RC-READ and RE-READNULL contain the notation `[`**`unique`**`]`, signifying that the rule applies whether or not the read is unique.

$$\frac{\mathbf{lent}\ \ldots \notin \overline{\mathtt{T}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}}{\mathbf{class}\ \mathtt{C}<\overline{\alpha},\overline{\beta}>\ \mathbf{extends}\ \mathtt{D}<\overline{\alpha}>\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \qquad (\text{T-CLASS})$$

$$\frac{\begin{array}{c}\overline{\mathtt{x}}:\overline{\mathtt{T}},\ \mathtt{this}:\mathtt{T}_\mathtt{this},\ \varnothing \vdash \mathtt{e}:\mathtt{T}' \qquad \mathtt{T}'\ <:\ \mathtt{T} \\ \mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha}>\ \mathbf{extends}\ \mathtt{D}\ldots \\ \mathit{override}(\mathtt{m},\ \mathtt{D},\ \mathtt{T}_\mathtt{this} \times \overline{\mathtt{T}} \to \mathtt{T}) \qquad \mathtt{T}_\mathtt{this} = \mathtt{A}\ \mathtt{C}<\overline{\alpha}> \\ \forall \mathtt{x} \in \{\overline{\mathtt{x}}, \mathbf{this}\}\ \mathbf{unique}(\mathtt{x})\ \textit{occurs at most once in}\ \mathtt{e}\end{array}}{\mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \mathtt{T}_\mathtt{this}\ \{\ \mathbf{return}\ \mathtt{e};\ \}\ \mathtt{OK\ in\ C}} \qquad (\text{T-METH})$$

$$\frac{\begin{array}{c}\mathit{dom}(\Sigma) = \mathit{dom}(\mathtt{S}) \\ \forall \ell \in \mathit{dom}(\mathtt{S})\ .\ \Gamma,\Sigma \vdash \mathtt{S}[\ell]\end{array}}{\Gamma,\Sigma \vdash \mathtt{S}} \qquad (\text{T-STORE})$$

$$\frac{\begin{array}{c}\mathit{CL}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha}>\ \ldots \qquad \Gamma,\Sigma \vdash \overline{\mathtt{v}} \in \overline{\mathtt{T}} \\ \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}_\mathtt{F}}\ \overline{\mathtt{f}} \qquad \overline{\mathtt{T}}\ <:\ [\overline{\ell}/\overline{\alpha}]\ \overline{\mathtt{T}_\mathtt{F}}\end{array}}{\Gamma,\Sigma \vdash \mathtt{C}<\overline{\ell}>(\overline{\mathtt{v}})} \qquad (\text{T-STORELOC})$$

$$\frac{\begin{array}{c}\mathtt{S}[\ell] = \mathtt{C}<\overline{\ell}>(\overline{\mathtt{v}}) \qquad \mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha}> \\ \mathit{fields}(\mathtt{C}) = \overline{\mathtt{A}}\ \overline{\mathtt{D}<\ldots>}\end{array}}{\mathit{annotation}(\mathtt{S},\ell,i) = [\overline{\ell}/\overline{\alpha}, \ell/\mathbf{owned}]\mathtt{A}_i} \qquad (\text{STOREANNOT})$$

Figure 15. AliasFJ Class, Method, and Store Typing

Typing Rules. Typing judgments, shown in Figure 14, are of the form $\Gamma,\Sigma \vdash \mathtt{e}:\mathtt{T}$, read "In the type environment Γ and store typing Σ, expression `e` has type `T`." The T-VAR and T-UVAR rules look up the type of a variable in Γ, replacing **`unique`** annotations with **`lent`** if the expression is not a unique variable read. Similarly, the T-LOC rule looks up the type of a location in Σ, leaving its annotation as expressed in the source text.

There are also two typing rules for field reads—the normal rule, which replaces **`unique`** annotations with **`lent`**, and the unique read rule, which leaves **`unique`** annotations unchanged. The rules for field read, field write, and method invocation verify that an **`owned`** value can only be accessed through the receiver `this` in the source text (naturally, reduction can replace `this` with a location).

Several of the typing rules use the auxiliary function *inst* (defined in Figure 16), which uses the type of the receiver of a method invocation or field access to convert the formal annotation variables referenced in the method or field type to the actual annotation variables used at the call site.

We have made one significant simplification relative to FJ. We do not distinguish between upcasts, downcasts, and so-called "stupid casts" which cast one type to an unrelated one. This means that our type system does not check for "stu pid casts" in the original typing derivation, as Java's type system does. However, the change shortens our presentation and proofs considerably, and the stupid casts technique from FJ can be easily applied to our system to get the same checks that are present in Java.

Store Typing. Figure 15 shows the rules for well-formed classes, methods, and stores in AliasFJ. Class and method typing rules check for well-formed class definitions, and have the form "dass declaration `E` is OK," and "method `m` is OK in `E`." The rules for class and method typing are similar to those in FJ. Rule T-CLASS ensures that subclasses can only extend the list of annotation parameters from their superclasses, and verifies that **`lent`** does

Field lookup:

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha},\overline{\beta}>\ \mathbf{extends}\ \mathtt{D}<\overline{\alpha}>\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{T}_\mathtt{g}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}_\mathtt{g}}\ \overline{\mathtt{g}},\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{\mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha},\overline{\beta}>\ \mathbf{extends}\ \mathtt{D}<\overline{\alpha}>\ \{\overline{\mathtt{T}_\mathtt{F}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{T\ m\ }(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \mathtt{T}_\mathtt{this}\ \{\ \mathbf{return}\ \mathtt{e};\ \} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathtt{T}_\mathtt{this} \times \overline{\mathtt{T}} \rightarrow \mathtt{T}}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha},\overline{\beta}>\ \mathbf{extends}\ \mathtt{D}<\overline{\alpha}>\ \{\overline{\mathtt{T}_\mathtt{F}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{m}\ \text{is not defined in}\ \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathtt{D})}$$

Method body lookup:

$$\frac{\mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha},\overline{\beta}>\ \mathbf{extends}\ \mathtt{D}<\overline{\alpha}>\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{T\ m\ }(\overline{\mathtt{T}_\mathtt{x}}\ \overline{\mathtt{x}})\ \mathtt{T}_\mathtt{this}\ \{\ \mathbf{return}\ \mathtt{e};\ \} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e})}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}<\overline{\alpha},\overline{\beta}>\ \mathbf{extends}\ \mathtt{D}<\overline{\alpha}>\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \qquad \mathtt{m}\ \text{is not defined in}\ \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \mathit{mbody}(\mathtt{m},\mathtt{D})}$$

Alias type instantiation:

$$\frac{\mathit{CT}(\mathtt{D}) = \mathbf{class}\ \mathtt{D}<\overline{p}>\mathtt{...} \qquad \mathtt{e}\ \mathit{not\ a\ location}}{\mathit{inst}(\mathtt{T},\mathtt{B\ D}<\overline{q}>,\mathtt{e}) = [\overline{q}/\overline{p}]\mathtt{T}}$$

$$\frac{\mathit{CT}(\mathtt{D}) = \mathbf{class}\ \mathtt{D}<\overline{p}>\mathtt{...}}{\mathit{inst}(\mathtt{T},\mathtt{B\ D}<\overline{q}>,\ell) = [\overline{q}/\overline{p},\ \ell/\mathbf{owned}]\mathtt{T}}$$

Valid method overriding:

$$\frac{\mathit{mtype}(\mathtt{m},\ \mathtt{C}) = \mathtt{T}_\mathtt{this} \times \overline{\mathtt{T}} \rightarrow \mathtt{T}_0 \Rightarrow \quad \overline{\mathtt{U}} = \overline{\mathtt{T}} \wedge \mathtt{U}_0 = \mathtt{T}_0 \wedge \mathtt{U}_\mathtt{this} = \mathtt{A\ C}_\mathtt{U}<\overline{\alpha},\overline{\beta}> \wedge \mathtt{T}_\mathtt{this} = \mathtt{A\ C}_\mathtt{T}<\overline{\alpha}>}{\mathit{override}(\mathtt{m},\ \mathtt{C},\ \mathtt{U}_\mathtt{this} \times \overline{\mathtt{U}} \rightarrow \mathtt{U}_0)}$$

Figure 16. AliasFJ Auxiliary Definitions

not appear in field types. Rule T-METH performs several checks. It ensures that the body is well typed in the environment that assumes the method arguments have their declared types, and an empty store. The rule also verifies that there is at most one unique read of each method argument (including `this`). Finally, the *override* auxiliary function verifies that each overriding method have the same type signature as the overridden method.

The store typing rules ensure that the form of the store is consistent with the Java's typing rules. The two clau ses of the store typing rule are the usual well-formedness rules, requiring the store type Σ to type every location in $\mathtt{S}$, and verifying that the types of objects in a field are compatible with the field's type using the auxiliary rule T-STORELOC. The last rule defines the *annotation* convenience function, which is used in stating the properties of the alias annotation system.

Auxiliary Definitions. Most of the auxiliary definitions shown in Figure 16 are straightforward and are derived from FJ. The field lookup rule returns the list of fields in a given class, along with their types. AliasFJ follows Java's lookup rules for method types and method bodies. The *inst* function accepts a type in a method or field signature as well as the type of the receiver of a method or field access, and converts the first type from its original scope to the scope of the method or field access. It does this by simply replacing the formal alias parameters in the signature type with the corresponding actual alias parameters in the receiver type. Finally, the last rule checks that overriding methods have the same type signatures as the methods they override, except that the class of `this` may differ.

3.2. Type Soundness

We can show the type soundness of AliasFJ through two standard theorems, subject reduction and progress. Type soundness implies that the language's type system is well behaved. In a type-safe language like Java, well-typed programs won't halt with errors due to calling a method on a class that doesn't define that method.

Theorem [Subject Reduction]: If $\Gamma,\Sigma \vdash \mathtt{e}:\mathtt{T}$, $\Gamma,\Sigma \vdash \mathtt{S}$ and $\mathtt{S} \vdash \mathtt{e} \rightarrow \mathtt{e}',\mathtt{S}'$, then $\exists \Sigma' \supseteq \Sigma,\ \mathtt{T}' <: \mathtt{T}$ such that $\Gamma,\Sigma' \vdash \mathtt{e}':\mathtt{T}'$ and $\Gamma,\Sigma' \vdash \mathtt{S}'$.

Before proving the theorem, we define a term substitution lemma, necessary for the method invocation case in the proof. This enables us to show that substituting terms in a well-typed expression preserves the typing:

Lemma [Term Substitution]: If $\overline{\mathtt{x}}:\overline{\mathtt{T}},\ \mathtt{this}:\mathtt{T}_\mathtt{this},\ \varnothing \vdash \mathtt{e}:\mathtt{T}$,

$\varnothing,\Sigma \vdash \overline{\mathtt{v}}:\overline{\mathtt{U}}$, $\varnothing,\Sigma \vdash \ell:\mathtt{U}_\mathtt{this}$, $\overline{\mathtt{U}} <: [\ell/\mathbf{owned},\ \overline{\ell}/\overline{\alpha}]\overline{\mathtt{T}}$,

$\mathtt{U}_\mathtt{this} <: [\ell/\mathbf{owned},\ \overline{\ell}/\overline{\alpha}]\mathtt{T}_\mathtt{this}$, $\overline{\mathtt{v}_\mathtt{lent}} = [\mathbf{lent}/\mathbf{unique}]\ \overline{\mathtt{v}}$, and

$\overline{\mathbf{this}_\mathtt{lent}} = [\mathbf{lent}/\mathbf{unique}]\ \overline{\mathbf{this}}$, then

$\varnothing,\Sigma \vdash [\overline{\mathtt{v}}/\overline{\mathbf{unique}(\mathtt{x})},\ \mathtt{A}_0(\ell)/\mathbf{unique}(\mathbf{this}),\ \overline{\ell}/\overline{\alpha},\ \ell/\mathbf{owned},$

$\overline{\mathtt{v}_\mathtt{lent}}/\overline{\mathtt{x}},\ \mathbf{this}_\mathtt{lent}/\mathbf{this}]\ \mathtt{e}:\mathtt{T}'$

for some $\mathtt{T}' <: [\ell/\mathbf{owned},\ \overline{\ell}/\overline{\alpha}]\mathtt{T}$.

The proof is by induction over the structure of $\mathtt{e}$, with a case analysis on the form of the outermost term:

Case T-VAR: The variable $\mathtt{x}$ must be one of **this** or $\overline{\mathtt{x}}$. But these variables were replaced with locations that are given a subtype of the variable types under the alias parameter mapping, so the case holds.

Case T-UVAR: As with T-VAR, but note that the rules preserve uniqueness.

Case T-LOC: The location is not affected by the substitution.

Case T-NEW: The substitution could affect the alias parameters of the new expression, but since the alias parameter substitution is included in the required subtype relation, the case holds.

Case T-CAST: Similar to T-NEW.

Cases T-ERROR, T-NULL: Unaffected by the substitution.

Cases T-FIELD, T-UFIELD, T-WFIELD: By the induction hypothesis, the receiver of the field access is a subtype of its original type under the alias parameter substitution. The instantiation of the field type with respect to this new receiver type must therefore also be a subtype of access expression's original type, so the cases hold.

Case T-INVK: Similar to T-FIELD. Here the induction hypothesis is used for the receiver and the method arguments. □

We then prove subject reduction by induction on the derivation of $S \vdash e \rightarrow e', S'$ with a case analysis on the outermost reduction rule used (one regular or error reduction rule may apply, in addition to any number of congruence rules).

Case R-NEW: We extend the store type to give ℓ the type $C\langle\bar{\ell}\rangle$, preserving the type of the expression. We know that the store will remain well-typed because we extend the store type's domain just as the store's domain is extended, and because the fields of the newly allocated object are initially null.

Cases R-READ, R-UNIQUEREAD: These cases follow from the original store typing and the rule T-FIELD.

Case R-WRITE: The expression type is unchanged. We know the store remains well-typed because T-WFIELD guarantees that the value to be stored is legal to put into the field.

Case R-CAST: The cast will only succeed if the resulting expression is a subtype of the cast type, so the case holds.

Case R-INVK: By simultaneous induction over the operation of *mtype* and *mbody*, we can see that the actual method has the type attributed by *mtype*. By applying the term-substitution lemma and the rule for well-typed methods, we can see that the substituted method body has a type that is a subtype of the original method call expression.

Subject reduction for the error rules follows since these rules reduce to the **error** expression, which has a type that is a subtype of all other types. Subject reduction follows for the congruence rules by applying the induction hypothesis, and noting that all of the typing rules hold when a subexpression is replaced with an expression that is a subtype of the original subexpression. □

Theorem [Progress]: If $\emptyset, \Sigma \vdash e : T$, then either e is an irreducible value, or e contains an **error** subexpression, or else $\forall S$ such that $\emptyset, \Sigma \vdash S$, $S \vdash e \rightarrow e', S'$.

The proof is by induction on the derivation of $\emptyset, \Sigma \vdash e \in T$, with a case analysis on the last typing rule used:

Cases T-VAR, T-UVAR: Impossible since we have the empty type environment.

Cases T-LOC, T-NULL: The expression is an irreducible value.

Case T-NEW: Reduction R-NEW applies.

Case T-CAST: If the cast expression is not a value, then either RC-CAST applies or there is an **error** subexpression. Otherwise, either R-CAST, R-CASTNULL, or RE-CASTFAIL must apply.

Case T-ERROR: The expression is **error**.

Cases T-FIELD, T-UFIELD: If the receiver expression is not a value, RC-FIELD applies, unless the receiver has an **error** subexpression. If the receiver is **null**, RE-READNULL applies. Otherwise, the induction hypothesis (together with the definition of *fields*) implies that the location ℓ refers to an object that has the field being read, and so either R-READ or R-UNIQUEREAD applies.

Case T-WFIELD: Similar to T-FIELD.

Case T-INVK: Similar to T-FIELD in the case of a receiver that is **null** or is not a value. In the case of a location as the receiver, the induction hypothesis implies that *mtype* returns a method type based on the type of the receiver. By simultaneous induction on the execution of *mtype* and *mbody*, we can see that *mbody* returns a method body, and so R-INVK applies. □

3.3. Properties

Type soundness is important, but we would also like to show that our system has well-defined properties that allow programmers to reason effectively about aliasing relationships. The first theorem gives the meaning of uniqueness: a **unique** annotation on a reference implies that no other heap references refer to that location.

Theorem [Uniqueness]: If $\emptyset, \emptyset \vdash e : T$ and $\emptyset \vdash e \rightarrow^{*} e', S'$, then for all ℓ such that ℓ occurs in S' or e' with annotation **unique**, all other occurrences of ℓ in S' or e' have annotation **lent**.

Formally, we say that ℓ occurs in S with annotation A if there exists some ℓ', i such that $S[\ell', i] = \ell$ and $annotation(S, \ell', i) = A$. We say that ℓ occurs in e with annotation A if $A(\ell)$ is a subexpression of e. Different occurrences are distinguished in the obvious way—by a pair (ℓ', i) for stores, and by textual location for expressions.

The crux of the proof is showing that the reduction rules obey three local properties: no duplication of unique references except with lent annotations, no flow from lent references to references with other annotations, and that whenever a unique reference flows to a reference with an ownership annotation, the original reference is dead. The proof is by induction on the derivation of $\emptyset \vdash e \rightarrow^{*} e', S'$, with a case analysis on the last reduction rule used:

Case R-NEW: ℓ occurs in the resulting expression with annotation **unique**, but there are no other occurrences of ℓ in the expression or in the store. The store is unchanged except for the newly created object ℓ, which cannot be referenced from the store, so the case holds due to the induction hypothesis.

Case R-READ: Assuming the induction hypothesis, the only way the uniqueness invariant could be violated is if the field was **unique** and its value was copied. However, if the field was **unique**, the value will be annotated **lent**, so the property holds in this case.

Case R-UNIQUEREAD: The induction hypothesis implies that the field holds the only **unique**-annotated reference to the location being read. Since the rule sets the field value to **null**, the read location will then be the only **unique**-annotated reference to that location.

Case R-WRITE: If the field is annotated **unique**, T-WFIELD ensures that the value was annotated **unique** as well. Since that value was formerly the only **unique**-annotated reference to its location by the induction hypothesis, and since the original value is eliminated by the reduction rule, the property holds in this case.

Case R-Cast: Does not affect the property, so it remains true by the induction hypothesis.

Case R-Invk: The main challenge here is to show that any **`unique`** arguments, possibly including **`this`**, are not duplicated when they are substituted into the method body. The crucial check, in rule T-Meth, ensures that a **`unique`** argument occurs at most once in the method body with a **`unique`** annotation. The substitution may copy the **`unique`** argument into this one place, but all other occurrences of that **`unique`** argument will be annotated **`lent`**.

The uniqueness property is also preserved the error rules and the congruence rules by applying the induction hypothesis, and noting that these rules do not perform any duplication of expressions that could lead to a violation of the invariant. □

We have argued in section 2.2 that ownership annotations are useful because they organize aliased objects into a hierarchical tree, and a group of objects can be persistently shared only if the group owner uses parameterization to delegate a capability to access the group. Intuitively, an object can only refer to an object if it has a capability to access that object. In order to allow this kind of reasoning about object ownership, we need the ownership annotations for an object to be consistent across the program's store and execution:

Theorem [Ownership Consistency]: If $\emptyset,\emptyset \vdash e : T$ and $\emptyset \vdash e \rightarrow^* e', S'$, then for all ℓ,ℓ' such that ℓ occurs in S' or e' with annotation ℓ', all other occurrences of ℓ in S' or e' have either annotation **`lent`** or annotation ℓ'.

The proof is by induction on the derivation of $\emptyset \vdash e \rightarrow^* e', S'$, with a case analysis on the last reduction rule used. The proof relies on the uniqueness property to show the base case: when a location is first given an owner, there is only one reference to that location. Once this is established, it is easy to show that the rules preserve ownership consistency:

Case R-New: The newly created location has annotation **`unique`**, and the store is unchanged except for the newly created object ℓ, which cannot be referenced from the store, so the case holds due to the induction hypothesis.

Case R-Read: This rule annotates the result of the read with the same annotation that the value had in the store (except replacing **`unique`** with **`lent`**). Thus, if the original value had an owner annotated in the store, it will remain annotated by the same owner. Thus, the property holds by the induction hypothesis.

Case R-UniqueRead: The read value is **`unique`**, and the uniqueness invariant implies that there are no owned aliases to it. Thus, this rule cannot violate the property.

Cases R-Write: The rule T-WField ensures that right and left sides of the assignment have compatible types. If the value is owned, the field must be owned by the same object, and so the invariant holds. If the value is **`unique`**, then by the uniqueness invariant, there are no aliases to it except possibly **`lent`** aliases. Thus, if a **`unique`** value is assigned to an owned field, the field will be the only non-**`lent`** reference to that location, so the property holds.

Case R-Cast: Does not affect the property, so it remains true by the induction hypothesis.

Case R-Invk: The argument is similar to the proof of subject reduction. We must also consider **`unique`** actuals that are passed to formals with an ownership annotation; this case is similar to the analogous situation in R-Write. □

Corollary [Ownership Soundness]: If $\emptyset,\emptyset \vdash e : T$, $\emptyset \vdash e \rightarrow^* e', S'$ and $S' \vdash e' \rightarrow^* e'', S''$, then for all ℓ,ℓ' such that ℓ occurs in S' or e' with annotation ℓ', all occurrences of ℓ in S'' or e'' have either annotation **`lent`** or annotation ℓ'.

The proof is similar. □

4. Annotation Inference

Although AliasJava is intended to give programmers the flexibility to express a wide variety of data sharing idioms, there are practical issues that may limit its adoption. In particular, adding alias annotations to existing programs and libraries may require significant work.

We have addressed this issue by developing a technique for inferring the annotations in AliasJava. The inference algorithm allows developers to easily infer the sharing relationships in library code or in legacy systems. If desired, programmers can refine the inferred declarations in order to enforce additional restrictions on aliasing.

Our 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.

4.1. 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`**: values that are returned from a method or assigned to a field. We also conservatively assume that the arguments of native methods are **`non-lent`**.

Next, our 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`**.

4.2. Inferring Unique

Our 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`**, except for the arguments and results of native methods.

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. We assume that live variable analysis has already annotated all value flows as ordinary assignment or last assignments.

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`**.

The graphs generated for both lent and unique inference are linear in the size of the source text, and traversing them touches each edge in the graph at most once. Therefore, our algorithm for inferring these alias types is linear in the size of the program.

4.3. 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. Type inference with instantiation constraints was first described in an abstract form by Henglein [Hen93]. More recent papers describe concrete worklist-based algorithms, which we have adopted in our work [FRD00,OCa00]. The underlying problem of finding an optimal solution for a set of component and instantiation constraints is undecidable [KTU93], and we have no proof that our inference algorithm terminates. However, in practice our algorithm works well; neither we nor others working on similar algorithms have ever encountered an example that causes the algorithm to loop [Hen93,FRD00,OCa00].

Our analysis is most similar to that used by O'Callahan in the Ajax system [OCa00]. O'Callahan's analysis can infer polymorphic types for static methods only. While our current analysis does not infer polymorphic types for methods, the type system supports them and we believe our analysis could be extended to infer these types for both static and instance methods. O'Callahan distinguishes different instances of a class based on their creation site, while our analysis distinguishes instances based on how they are used in the system. Thus, we are able to distinguish different objects that are created at the same place but are used in different ways, but we don't waste effort tracking objects that are created in different places but are used in the same way.

We first present a high-level overview of the algorithm in parallel with an example that illustrates many of the key issues, then give the formal constraint generation rules and constraint solution algorithm. We choose as our running example the `Stack` code in Figure 3, 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 3. 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 17(a) shows the type variables generated from Figure 3. 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.

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 `member`, we don't equate the corresponding *pop* and *member* variables, because that would place unnecessary constraints on other parts of the program that use `Link.member`. We use instantiation constraints (discussed below) to reason about value flow between classes in a way that treats different `Link` objects differently. Figure 17(b) shows the equality constraints generated from Figure 3.

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 17 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 ($o \triangleright_m v$), 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 $C \triangleright_m m$. 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 17(b) shows the component constraints generated from Figure 3.

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

(a) Initial variables:

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

(b) Initial constraints:

Equality:

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

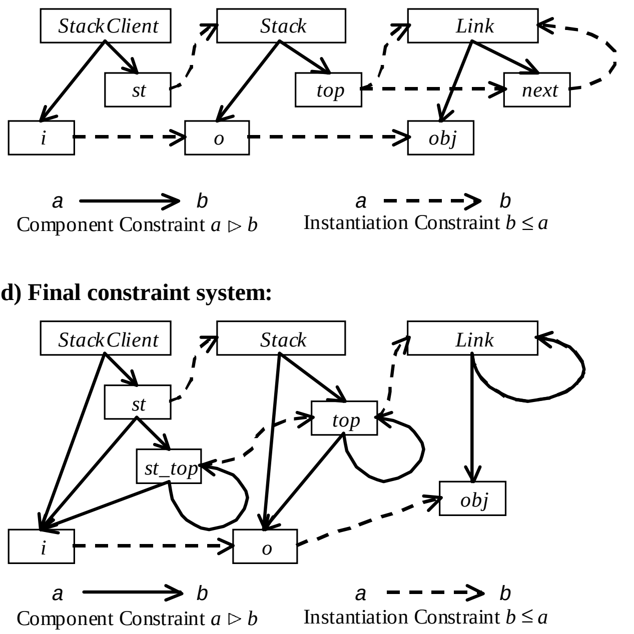
Component:

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

Instantiation:

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

(c) After solving initial equality & uniqueness constraints:


StackClient
Stack
Link
st
top
next
i
o
obj
a
b
Component Constraint a ▷ b
a
b
Instantiation Constraint b ≤ a
StackClient
Stack
Link
st
top
st_top
obj
i
o
a
b
Component Constraint a ▷ b
a
b
Instantiation Constraint b ≤ a

(d) Final constraint system:

Figure 17. Constraints generated and solved during inference of the alias types given in Figure 3.

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 17(b) shows the instantiation constraints generated from Figure 3.

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 *member* 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:

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

Figure 17(c) shows the example system after solving the initial equality constraints and applying the uniqueness rules.

Constraint Generation Rules. Figure 18 presents the formal constraint generation rules for unannotated AliasFJ code. The rules are syntax-directed, and are based on the formal language defined in section 4 with all alias annotations and parameters erased. In the rules, the notation `[e]` denotes the type variable for the expression `e`. A constraint generation rule is of the form `C,Γ ⊢ e / X`, read "In the context of class `C` and type environment Γ, the constraints `X` are generated as a side effect for expression `e`" The initial system of constraints is defined as the union of the constraints generated for every class in the system.

The rules for variables and for null generate no constraints. The field read and write rules constrain the type variable for the read or written expression to be an instance of the type variable for the field in the static class of the receiver, labeling the constraint with the type variable of the receiver expression. The rule for new constrains the type variable for the created object to be an instance of the type variable for the object's class; the label is the created object's type variable. A cast constrains the type variable for the cast expression to be an instance of the type variable for the cast class–we do not model the failure of the cast, since our analysis is flow-insensitive.

The method invocation rule constrains the method invocation expression's variable to be an instance of the method's return type, and constrains the actual arguments and receiver to be instances of the formal method arguments and **`this`**. The field and method declaration rules state that the type variables for the

$$\mathtt{C},\Gamma \vdash \mathtt{x} \ / \ \varnothing \qquad (\text{I-VAR})$$

$$\mathtt{C},\Gamma \vdash \mathtt{null} \ / \ \varnothing \qquad (\text{I-NULL})$$

$$\frac{\Gamma,\varnothing \vdash \mathtt{e}:\mathtt{A}\ \mathtt{D}<\overline{\alpha}>}{\mathtt{C},\Gamma \vdash \mathtt{e.f} \ / \ [\mathtt{f}_\mathtt{D}]\leq_{[\mathtt{e}]}[\mathtt{e.f}]} \qquad (\text{I-FIELDREAD})$$

$$\frac{\Gamma,\varnothing \vdash \mathtt{e}_1:\mathtt{A}\ \mathtt{D}<\overline{\alpha}>}{\begin{array}{c}\mathtt{C},\Gamma \vdash \mathtt{e}_1\mathtt{.f}=\mathtt{e}_2\mathtt{;}\ \mathtt{e}_3 \ / \\ [\mathtt{f}_\mathtt{D}]\leq_{[\mathtt{e}_1]}[\mathtt{e}_2] \ \cup \ [\mathtt{e}_1\mathtt{.f}]=[\mathtt{e}_2]\end{array}} \qquad (\text{I-FIELDWRITE})$$

$$\mathtt{C},\Gamma \vdash \mathtt{new\ D()} \ / \ [\mathtt{D}]\leq_{[\mathtt{new\ D()}]}[\mathtt{new\ D()}] \qquad (\text{I-NEW})$$

$$\mathtt{C},\Gamma \vdash \mathtt{(D)e} \ / \ [\mathtt{(D)e}]=[\mathtt{e}] \ \cup \ [\mathtt{D}]\leq_{[\mathtt{e}]}[\mathtt{e}] \qquad (\text{I-CAST})$$

$$\frac{\Gamma,\varnothing \vdash \mathtt{e}_0:\mathtt{A}\ \mathtt{D}<\overline{\alpha}> \qquad \mathit{mbody}(\mathtt{m},\mathtt{D})=(\overline{\mathtt{x}_\mathtt{D}},\mathtt{e}_\mathtt{m})}{\begin{array}{c}\mathtt{C},\Gamma \vdash \mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}}) \ / \ [\mathtt{m}_\mathtt{D}]\leq_{[\mathtt{e}_0]}[\mathtt{e}_0\mathtt{.m}(\overline{\mathtt{e}})] \\ \cup \ [\mathtt{D}]\leq_{[\mathtt{e}_0]}[\mathtt{e}_0] \ \cup \ \overline{[\mathtt{x}_\mathtt{D}]\leq_{[\mathtt{e}_0]}[\mathtt{e}]}\end{array}} \qquad (\text{I-INVK})$$

$$\mathtt{C},\Gamma \vdash \mathtt{D\ f} \ / \ [\mathtt{C}]\rhd_{[\mathtt{f}]}[\mathtt{f}_\mathtt{C}] \ \cup \ [\mathtt{D}]\leq_{[\mathtt{f}]}[\mathtt{f}_\mathtt{C}] \qquad (\text{I-FIELDDECL})$$

$$\begin{array}{c}\mathtt{C} \vdash \mathtt{D\ m}(\overline{\mathtt{E}}\ \overline{\mathtt{x}})\ \mathtt{\{\ \ return\ e;\ \}} \ / \\ [\mathtt{C}]\rhd_{[\mathtt{m}]}[\mathtt{m}_\mathtt{C}] \ \cup \ [\mathtt{D}]\leq_{[\mathtt{m}]}[\mathtt{m}_\mathtt{C}] \ \cup \ [\mathtt{m}_\mathtt{C}]=[\mathtt{e}] \\ \cup \ \overline{[\mathtt{C}]\rhd_{[\mathtt{x}]}[\mathtt{x}_\mathtt{C}]} \ \cup \ \overline{[\mathtt{E}]\leq_{[\mathtt{x}]}[\mathtt{x}_\mathtt{C}]}\end{array} \qquad (\text{I-METHDECL})$$

$$\mathtt{class\ C\ \ extends\ D\ \{}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}} \ / \ [\mathtt{D}]\leq_{[\mathtt{C}]}[\mathtt{C}] \qquad (\text{I-CLASS})$$

Figure 18. Constraint generation rules for parameter inference

field and for the method arguments and result are components of the surrounding type, and constrain them to be instances of the variables for their declared types. Finally, the class rule states that the type variable for a class is an instance of the type variable for its superclass.

Note that each class C has its own type variable for each field f, denoted $\mathtt{f}_\mathtt{C}$ (and similarly for method arguments and results). Because $\mathtt{f}_\mathtt{C}$ is a component of C with a label f that is not dependent on the class, constraint propagation (discussed below) will ensure that when D is C's superclass, then $\mathtt{f}_\mathtt{C}$ will be an instance of $\mathtt{f}_\mathtt{D}$ with label C. This establishes the required inheritance relationship between the fields of a supertype and the corresponding fields of a subtype.

Constraint Propagation. If top is an instance of Link, as shown in Figure 17(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:

$$a \rhd_b c \wedge a \leq_I d \ \Rightarrow \ \exists\, e \,.\, d \rhd_b e$$

Applied to *top*, this rule states that since *Link* has a component *next* ($\mathit{Link} \rhd_{next} \mathit{next}$) and *top* is an instance of *Link* ($\mathit{Link} \leq_{top} \mathit{top}$), then there must exist some variable *top_next* such that *top_next* is a component of *top* at index *next* ($\mathit{top} \rhd_{next} \mathit{top\_next}$). 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:

$$a \rhd_b c \wedge a \leq_I d \wedge d \rhd_b e \ \Rightarrow \ c \leq_I e$$

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 $\mathit{next} \leq_{top} \mathit{top\_next}$.

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.

Like O'Callahan and others, 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 $\exists\, L \rhd_{c1} b_1 \rhd_{c2} \ldots \rhd_{cM} R$
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 $\mathit{Link} \rhd_{next} \mathit{next} \wedge \mathit{Link} \leq_{next} \mathit{next}$. Thus, our implementation generates the equality constraint *next*=*Link*, which eliminates the source of the loop.

Figure 17(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 3. *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.

Constraint Solution Algorithm. Figure 19 presents our constraint solution algorithm. The constraints are stored in two data structures: cmap for component constraints, and imap for instantiation constraints. The data structures are indexed to allow quick lookup based on the left-hand side, right-hand side, and index of each constraint.

We solve the constraints using a iterative worklist algorithm. We initialize the worklist with the initial set of constraints, and then we add them one by one to the data structures. For efficiency, the fetch function returns first equality constraints, then component

```
solve(initial_constraints)
        Wlist := initial_constraints
        cmap := $\emptyset$ // stores component constraints:  $a \blacktriangleright_b c$      $\Rightarrow$ cmap[a][b] = c
        imap := $\emptyset$ // stores instance constraints:     $a \leq_b c$      $\Rightarrow$ imap[a][b] = c
        dirty := $\emptyset$ // stores instance constraints that might need propagation rules applied

        while (Wlist not empty $\vee$ dirty not empty) do
            if (Wlist not empty)
              add(fetch(Wlist))
            else if (dirty not empty)
              propagate(fetch(dirty))

fetch(Wlist)  // returns equalities, followed by component and then instantiation constraints

add(rep = other)
        if (find(rep) = find(other)) return;
        ecr(other) := rep;    // perform Tarjan's union operation
          // keep the constraint maps up-to-date, where the hash table keys have been unified
        for all $a \blacktriangleright_c b \in$ cmap where a = other or c = other
                remove $a \blacktriangleright_c b$ from cmap and add $a \blacktriangleright_c b$ to Wlist
        for all $a \leq_c b \in$ imap where a = other or c = other
                remove $a \leq_c b$ from imap and add $a \leq_c b$ to Wlist
        for all $a \leq_c b \in$ dirty where a = other or c = other
                remove $a \leq_c b$ from dirty, and add find(a) $\leq_{\text{find(c)}}$ find(b) to dirty

add($L \blacktriangleright_c R$)
        L := find(L); c = find(c); R = find(R);
        if cmap[L][c] = null
                cmap[L][c] := R
                dirty = dirty $\cup$ { $L \leq_I b \mid L \leq_I b \in$ imap }
        else if $L \blacktriangleright_c b \in$ cmap $\wedge$ find(b) $\neq$ R
                Wlist = Wlist $\cup$ { R = b }

add($L \leq_I R$)
        L := find(L); I = find(I); R = find(R);
        if imap[L][I] = null
                imap[L][I] := R
                EOC($L \leq_I R$) // extended occurs check
                dirty = dirty $\cup$ { $L \leq_I R$ }
        else if $L \leq_I b \in$ imap[L][I] $\wedge$ find(b) $\neq$ R
                Wlist = Wlist $\cup$ { R = b }

propagate($L \leq_I R$)         // implements the constraint propagation rules
        $\forall\ L \blacktriangleright_c v \in$ cmap
                if (cmap[R][c] = null && imap[v][I] = null)
                        w := fresh
                        Wlist = Wlist $\cup$ { $R \blacktriangleright_c w$ } $\cup$ { $v \leq_I w$ }
                else
                        Wlist = Wlist $\cup$ { $v \leq_I w \mid R \blacktriangleright_c w \in$ cmap }
                        Wlist = Wlist $\cup$ { $R \blacktriangleright_c w \mid v \leq_I w \in$ imap }

EOC($L \leq_I R$)              // O'Callahan's extended occurs check
        if $\exists$ a such that a $\rightarrow^*$ L $\in$ imap and a $\rightarrow^*$ R $\in$ cmap
                Wlist = Wlist $\cup$ { a = R }
```

Figure 19. Constraint solution algorithm

constraints, then instantiation constraints; processing equality constraints first reduces the number of the constraints in the system as aggressively as possible.

The add function for equality constraints uses Tarjan's union -find algorithm to merge the equated nodes. Because imap, cmap, and dirty are indexed on the first two elements, we must update these data structures whenever one node is merged into another node. For example, if node *other* is merged into node *rep*, all of the component constraints that have *other* on the left-hand side must be removed from the cmap and added to the worklist, so they can be added with the *rep* node on the left-hand side.

The functions for adding component and instantiation constraints are similar. Ignore for a moment the updates to dirty and the calls to EOC. These functions check to see if there is already a constraint in the cmap or imap that has the same left-hand side and index as the constraint being added. If there is not, they add the new constraint to the map. If there is a pre-existing constraint, the functions add a new equality constraint between the right-hand sides of the old and new contraints. This implements the component and instance uniqueness rules.

While it is adding constraints, the solver must also keep the dirty list up to date. This list holds all of the instantiation constraints for which the propagation rule needs to be executed. Whenever a new instantiation constraint is added to imap, it is added to the dirty list. Likewise, whenever a new component constraint is added to cmap, all of the instantiation constraints that have the same left-hand side as the component constraint are dirty, since the new component needs to be propagated to the instances of the left-hand side.

When the worklist is empty, a constraint is removed from the dirty list and the propagation rule is applied. The propagate function implements this in a straightforward way. For each component of the left-hand side of the dirty instantiation constraint, the function checks whether a corresponding component of the right-hand side exists. If it does not, a fresh node is created and constraints are added representing the two propagation rules. If it does exist, but only one of the two required propagation constraints is there, the algorithm adds the missing constraint. The top-level solve function adds any new constraints to the system before proceeding to propagate the next dirty constraint.

Finally, whenever a new instantiation constraint is added to the system, the extended occurs check in function EOC is executed. The EOC function checks whether the newly added constraint causes one node to be both an instance and a component of another, and unifies the two nodes if this is the case. The cmap and imap data structures maintain an efficient backwards index (from the right-hand side to the left-hand side of a constraint) in order to implement the EOC check relatively efficiently. O'Callahan's thesis discusses more aggressive ways to optimize this check [OCa00].

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 wi ll 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.

Optimizations. The algorithm described above terminates efficiently in practice, but yields imprecise results. In particular, the aggressive merging caused by the EOC function mixes too many **`shared`** nodes with nodes that would otherwise be marked **`owned`** or an alias parameter. In fact, in our experience nearly all nodes become **`shared`**. Therefore, we have applied heuristics that avoid mixing **`shared`** nodes with unshared nodes.

Our technique tracks which nodes are **`shared`** online. Thus, whenever two nodes are merged, the resulting node is marked **`shared`** if either source node was **`shared`**. Similar checks are done whenever a component or instantiation constraint is added to the system: if the left-hand side is **`shared`**, then the right-hand side becomes **`shared`** as well.

We modify the EOC function to keep track of which nodes would be unified, without actually adding the resulting equality constraints to the worklist. We then divide the list into **`shared`** and unshared nodes, and unify these sets separately (but not with each other). This avoids much of the imprecision.

Another empirically-determined source of imprecision comes from merging the nodes representing different classes together. When one class becomes shared, the classes it is merged with follow, even if this is not semantically necessary. We sidestep this problem by checking to see if two or more class nodes are present in a set of nodes unified by the EOC function. If so, all the class nodes are removed from the set before unifying it. As a result, we went from a situation where nearly all class nodes were unified together, to a situation where a relatively small fraction were unified.

As a result of these optimizations, we have reduced the number of nodes determined to be **shared** substantially—from nearly 100% to below 50%. While there are still more **`shared`** nodes inferred than we would like, we believe that improvements to the encoding we use, such as supporting the inference of method alias parameters, will allow continued improvement in the precision of our algorithm.

5. Evaluation

A significant deficiency of previous work on specifying object ownership is that no significant experience has been reported regarding the usability of these systems in practice. We have evaluated AliasJava with three experiments. To test our system's flexibility on collection library code, we added alias annotations by hand to the `Hashtable` class from the `java.util` library. To determine if meaningful data-sharing relationships between components can be represented in a software architecture, we applied our system to Aphyds, the subject of a previous ArchJava case study [ACN02a]. Finally, we measured the effectiveness of annotation inference by comparing inference results to small hand-annotated examples, and measured its scalability by running part of it on over 400 classes from the Java standard library.

5.1. Hashtable

Motivation. Collection class code is a challenge for alias annotation systems, because collection classes and their iterators often store references to data objects that are logically a part of application objects. Collection classes were a significant part of the design motivation for Flexible Alias Protection. Thus, collection classes are an important test of any alias annotation system.

We have evaluated AliasJava by annotating `Hashtable` from the `java.util` collection class library (from the JDK 1.2.1). `Hashtable` is an interesting test case for a number of reasons. The class must distinguish different alias types for the keys, values, and possibly the entries in the `Hashtable`. `Hashtable` is also one of the more complex pieces of the library, so it is a relatively challenging test case. Finally, we wanted to test our system on an industrial-strength library with many features and warts. The Flexible Alias Protection paper used a simplified version of `Hashtable` as a running example in their paper, so this allows a partial comparison to related work [NVP98].

Goals. The goals of our study included answering the following experimental questions:

- Can the annotation system effectively express the aliasing invariants of collection class code?
- How much effort is required to annotate existing code?
- Can annotations be done locally, without annotating all transitively reachable code?

Methodology. The subject of our study was the source code to `java.util.Hashtable` from the JDK 1.2.1. The original source was 934 lines of code, including comments. We added alias annotations by hand to the `Hashtable` code, attempting to express the aliasing semantics of the code with the simplest and most flexible annotations possible.

In this study, we tested a local annotation technique intended to allow us to verify the alias constraints within the `Hashtable` code without annotating the entire Java standard library. We annotated and typechecked `Hashtable` in its entirety, but added only minimal, unchecked annotations to the parts of the standard library used by `Hashtable`. The annotations added to `Hashtable` are then sound if the annotations we added to the standard library are conservative.

Results. We were successful at annotating `Hashtable` with alias types after making one change to the source code (discussed below). In addition to modifying the code for `Hashtable`, partial annotations were added to 17 other classes, including `java.lang.Object`, `ObjectInputStream` and `ObjectOutputStream` from the I/O library, several interfaces and abstract classes in `java.util`, and seven exception classes. In most cases we only had to annotate one or two methods from each external class, suggesting that it is practical to annotate only a local portion of a large system.

The study took about 2 hours and 20 minutes of programming time, not counting occasional interruptions to fix problems with the compiler. This is a relatively small investment compared to the time spent developing this library, suggesting that our annotation system is practical for developing new code. However, it would still be time-consuming to add alias annotations to a very large system; a better solution is to infer the annotations automatically, or add annotations incrementally to just the most critical parts of the system.

Several excerpts from the source code highlight lessons learned from the study. For example, we decided to give `Hashtable` three parameters: one each for keys, values, and entries:

```
public class Hashtable<key, value, entry>
             extends Dictionary<key, value>
             implements Map <key, value, entry>,
                        Cloneable,
                        java.io.Serializable { ...
```

The choice of three parameters is a balance between flexibility on the one hand and simplicity and comprehensibility on the other. For example, we could have reduced the number of parameters by merging the `entry` and `key` parameters. On the other hand, we could have added additional parameters also. For example, `Hashtable` has methods for returning the sets of keys, values, and entries. We chose to annotate the `keySet` method's return type as `key Set<key>`, but we could have added extra alias parameters to `Hashtable` to get a type of `keyset`

Set<key>. However, adding three extra alias parameters to the hash table to represent the key, value, and entry sets would make the class harder to understand and use. This example illustrates that *the best alias annotation for a piece of code is not necessarily the most general*.

The private inner Enumerator class below is part of the original, unannotated code defining an Iterator over the keys, values, and entries of the Hashtable:

```
  private class Enumerator implements Iterator {
    int type; // KEYS or VALUES or ENTRIES
    public Object nextElement() {
      Entry e = ...;
      return type == KEYS ? e.key :
             (type == VALUES ? e.value : e);
    }
  }
```

The same code is used for keys, values, and entries; the value returned by nextElement is determined by the value of the type flag. Because we wanted to use separate alias parameters for keys, values, and entries, we could not give this code a static type as it was. Instead, we converted this code to always return an entry so that we could give it the alias type entry. We then defined two wrapper classes that implement Iterator and extract and return the key and value from the hash table entry returned by Enumerator.nextElement.

The set of Hashtable keys is implemented with a simple KeySet class that illustrates how inner classes are handled in our system:

```
  private class KeySet extends AbstractSet<key> {
    public unique Iterator<key> iterator() lent {
      return new KeyEnumerator(true);
    }
    // other methods...
  }
```

In this code, class Keyset can reference the key parameter of the enclosing Hashtable class even though KeySet has no alias parameters of its own.

The class Collections contains a set of static methods that are used by many of the classes in java.util:

```
public class Collections {
  public static unique Set<elements>
    synchronizedSet<elements>(
                         unique Set<elements> s) {
      return new SynchronizedSet(s);
  }
```

The synchronizedSet method is used by the Hashtable to synchronize access to its key, value, and entry sets. This method shows the need for method parameterization in our annotation system: synchronizedSet needs to be parameterized by the owner of the elements in the collection so that it can be used to synchronize sets with any element parameter.

The comment for the method above states, ‘In order to guarantee serial access, it is critical that **all** access to the backing set is accomplished through the returned set.” In other words, there should be no aliases to the set passed to this method, because access through these aliases would not be synchronized. The original library did not enforce this constraint; however, we used our alias annotation system to enforce this constraint by annotating the set argument with **unique**.

Problematic Classes. As described above, we annotated a number of other classes in addition to Hashtable; these annotations were not checked by the compiler, but Hashtable was checked against the asserted annotations. In general, the annotations we applied to classes other than Hashtable were what we would expect to have used if our compiler had been checking those annotations as well. The lone exceptions were certain methods of ObjectInputStream and ObjectOutputStream. Our annotation system expressed the conceptual semantics of these serialization-related methods (e.g., writeObject accepts a **lent** argument and readObject returns a **unique** object). However, the actual implementation of these methods caches object references in order to save and restore object graphs that contain sharing. Therefore, AliasJava would be unable to typecheck the implementations of these classes against these alias annotations. Although it would be nice to handle this example in our system, we can easily typecheck clients of these classes by asserting an alias annotation interface that expresses the desired semantics; we could also provide an unsound alias *annotation* cast to complement our system's existing, sound cast (which checks alias *parameters* at run time).

5.2. Aphyds

We wanted to evaluate AliasJava on application code as well as library code, in order to answer the following experimental questions:

- Is the annotation system practical on realistic application code?
- Does the annotation system help to encode application-specific architectural constraints?

Methodology. We performed a case study, adding alias annotations to the architecture of an existing ArchJava application. The subject of our study was Aphyds, a pedagogical circuit layout application written by an electrical engineering professor for one of his classes. Students are given the program with several key algorithms omitted, and are asked to code the algorithms as assignments. The source code is about 12,500 lines long.

In previous work, we expressed the control-flow architecture of Aphyds, as drawn by the developer, using the ArchJava language [ACN02a]. The intention of this study is to express the data sharing relationships in the architecture using the alias annotation system as an addition to ArchJava.

Aphyds has an architecture that follows the model-view design pattern [GHJ+94]. A set of user interface windows forms the view, and interacting with the model to execute circuit operations and display circuit elements. The model has an internal repository-style architecture, with a set of five computational components surrounding and interacting with a central data store of circuit elements.

In this study, we focused on the model part of Aphyds. Our goal was to express the data sharing relationships between the components in the architecture. Thus, we applied AliasJava to the AphydsModel class representing the overall model's architecture, as well as the Circuit repository and the five computational module classes. These 7 large classes comprise

3550 lines of code, as measured by Unix `wc` (word count). We typechecked the alias annotations in these classes against annotations we added to parts of the interfaces of the Java standard library and the rest of the Aphyds application.

Results. The study took about three hours and 40 minutes—less than a quarter of the time that it took the same programmer to express the control-flow architecture of the same part of Aphyds. The alias annotation system probably required editing more lines of source text than the earlier, control-flow architecture annotations. However, the alias annotations did not require changing any existing source code, just adding annotations. In contrast, our earlier system required significant source-code refactoring to make the code conform to the developer's intended architecture.

We discovered almost immediately that it was quite tedious to annotate the majority of method arguments (including **`this`**) and local variable declarations that have a **`lent`** annotation. We have since made **`lent`** the default annotation for method arguments and locals.

The annotations in the architecture show the style of sharing in this repository application. The circuit database has a single alias parameter, `data`, that represents the circuit elements in the database. Since all of the other computational components act on these circuit elements, they are also parameterized by the same alias parameter. We did not use the **`shared`** annotation except for objects of type `String`. `String` objects are immutable in Java, so we did not feel that it was important to track their aliasing patterns precisely, and making strings **`shared`** simplified our annotation task.

The annotations in ports used for communication between components also show the semantics of the methods used for inter-component communication. Methods that return computed data typically take **`lent`** parameters and return results annotated either **`unique`** or `data`. In contrast, methods that set data usually take parameters with `data` annotations. These annotations also showed that the objects shared between components came from a small set of classes including circuit elements and data structures that reflect their organization into a circuit.

5.3. Annotation Inference

We evaluated our annotation inference algorithm in several ways. First, we applied inference to small examples, and compared the inferred types with those generated by hand. We then evaluate the scalability of inference in time and space using the Java Standard Library. Finally, we report our observations on the inferred types for Java library code.

Inference Benchmarks. We chose as our inference benchmarks a set of code examples taken from this paper, specifically Figures 1 through 6. These examples do not involve ArchJava code (for which our annotation inference implementation is not yet complete). We ran the inference algorithm on versions of the code that had all annotations stripped out.

Our implementation of annotation inference inferred exactly the same types as are shown in the figures (up to renaming of parameters), with the following exceptions. We inferred **`lent`** or **`unique`** annotations for a few local variables that have **`owned`** or **`shared`** annotations in the figures (for example, `temp` and `i2` in Figure 3 and `s` in Figure 4 were **`lent`**, and the points in Figure 2 were **`unique`**). In this case, the annotations inferred by the inference algorithm were in fact more precise than the ones in the figures.

Scalability. Our inference algorithms for **`lent`** and **`unique`** scale linearly with program size. We timed the algorithms on the 408 classes in the JDK 1.2.1 standard library that are reachable from `java.lang.Object`. As a point of reference, it takes about 100 seconds for our compiler to parse and typecheck these classes. Our **`lent`** and **`unique`** inference analyses took 33 seconds and 151 seconds, respectively. Thus inferring these annotations takes time comparable to parsing and typechecking.

Our current constraint solver implementation has been partly optimized, but we will continue to improve execution time and space using techniques developed by O'Callahan and others [OCa00]. The solver infers alias parameters for the 408 Java standard library classes in about 30 minutes, using 2 GB of memory.

Standard Library Inference. We ran our inference algorithm on the same 408 classes from the Java Standard Library to determine whether the inferred annotations would be both precise and understandable.

Our experiment suggests that the inference algorithm is fairly precise, although some improvement is still needed. Around 50% of method and constructor parameters were inferred to be either **`lent`** or **`unique`**, which represent the most precise annotations. The other annotations were split about equally between **`shared`** and alias parameters, with a few **`owned`** annotations also.

The major symptom of imprecision in the inference results appears to be unnecessary **`shared`** annotations. We have found that inference results are very sensitive to the way that the type system is encoded into constraints, and the way that the extended occurs check unifies type variables. By experimenting with different constraint encodings and unification heuristics, we have been able to reduce shared annotations considerably, and we believe there is still room for improvement. We hope to eliminate another major source of imprecision by implementing inference of static method parameters.

When evaluating the understandability of the inferred annotations, we discovered that many classes had dozens if not hundreds of inferred parameters. In a sense, the analysis is too precise, making distinctions between different alias parameters that are too fine to be useful to the programmer. Our experience suggests that additional tools or heuristics will be needed to reduce the number of parameters for each class to a manageable level.

6. Related Work

Our work builds on a number of existing type systems for describing alias relationships in object-oriented programs. The most closely related work falls into two main categories: uniqueness type systems for describing unaliased pointers, and ownership type systems for describing pointers that are confined to a limited domain. AliasJava combines these lines of research, supporting both unique references and a flexible form of object ownership. The synergy of these features allows AliasJava to express important idioms that neither class of annotations can express alone, such as those discussed in section 2.4.

Uniqueness types can be used to declare references that are unaliased [Min96, CBS98]. Passing a unique object from one method to another avoids all aliasing problems, since the original method may not use the object again. Our *lent* annotation is similar to Wadler's **`let!`** Construct [Wad90]. Boyland's alias burying paper [Boy01] described how to implement unique pointers without a special destructive read operation, an innovation adopted by AliasJava. Alias burying uses an effect system to enforce a stronger uniqueness invariant than AliasJava enforces: namely that when a unique field is read, all previous lent aliases to that field are dead.

Linear type systems [Wad90] guarantee uniqueness and in addition can be used to track resource usage. Linear types have been applied to check protocols defining the order in which library methods can be called, as in the Vault language [FD02]. Leino et al. have also used uniqueness to specify and check side effects in a modular way [LHZ02]. A number of research efforts have used linear types to verify the correctness of explicit memory management using the concept of a region [TT94,CWM99,FD02,GMJ+02]. A region represents a group of objects that are deallocated together. A region type is similar to an ownership type in that all objects must be accessed through their region. Although supporting explicit deallocation is not a goal of AliasJava, our system makes two contributions relative to region types. First, regions must be tracked linearly to enable explicit deallocation; AliasJava relaxes this constraint on owning objects, permitting more flexible aliasing patterns. Second, region types do not have an encapsulation model like AliasJava's for protecting access to the objects in a region; any object that can name the region can access the objects inside it.

Ownership types, which describe a limited static or dynamic scope within which sharing can occur, can also be used to control aliasing. Early work such as Islands [Hog91] and Balloons [Alm97] imposed strict rules on sharing objects between components, significantly limiting expressiveness. A more recent variation, Confined Types [BV99], allows programmers to restrict object references to within a particular package; the system has been extended to support inference of confined types [GPV01]. Universes [MP99] provides a combination of ownership and confinement, providing additional flexibility using read-only references that can cross universe boundaries. More recently, Clarke et al. and Banerjee et al. have used ownership types to reason about side effects and representation independence as well as aliasing [CD02, BN02].

The ownership annotations in AliasJava are most closely related to Flexible Alias Protection [NVP98] and its successors [CPN98,CNP01,Cla01]. Flexible Alias Protection uses ownership polymorphism to strike a balance between guaranteeing aliasing properties and allowing flexible programming idioms. In Flexible Alias Protection, owned objects can only be accessed by their owner and its children. However, this invariant prohibits iterators, which are not owned by a collection, yet must access its owned state. Clarke *et al.* address this issue by introducing a new abstraction called ownership contexts: each object has an *owning context* (the context that owns it) and a *representation context* (the context that owns its representation) [CNP01, Cla01]. The key property of their system is a *containment invariant*, which states that if object o_1 refers to object o_2, then the representation context of o_1 must be *inside* the *owning* context of o_2.

The ownership subset of AliasJava is quite similar to that of Clarke's thesis [Cla01] in both expressiveness and the properties enforced. We wanted to enforce an encapsulation property that relates objects directly, rather than one that relates abstract ownership contexts. Therefore, we chose to phrase the encapsulation guarantees of AliasJava in terms of capabilities that can be passed from one object to another using ownership parameters. AliasJava's capability-based encapsulation is slightly weaker than Clarke's containment invariant because we place no restrictions on alias parameters, but AliasJava is correspondingly more flexible. Existing implementations of Flexible Alias Protection and its successors lack support for language features such as inheritance [Bok99, Buc00], and thus there has been no significant experimental validation of the design.

Capabilities for Sharing [BNR01] describes a general capability-based aliasing model that can encode a number of other alias-control systems, including ours, as a special case. The capabilities in their system are fine-grained and are dynamically checked; in contrast, our type system verifies statically (except for casts) that objects are only accessed through appropriate high-level capabilities.

Parameterized Race Free Java (PRFJ) uses the concept of object ownership and uniqueness to develop a type system to guarantee that a program is free of data races [BR01] and deadlocks [BR02]. PRFJ was not designed to encapsulate owned objects. However, a variant of PRFJ supports a stronger notion of object encapsulation than AliasJava: owned objects are confined within the owner, its owned objects, and its inner classes [BR02]. This variant is more restrictive than AliasJava: an object can delegate a capability to access its owned state to its other owned objects and to its inner classes, but not to trusted external classes and methods, even temporarily. Thus, iterators can only be implemented as inner classes of the collection they iterate over. Also, objects cannot be unique if they have non-shared, non-unique ownership parameters—prohibiting many uses of unique. To our knowledge, this variant has not been evaluated in practice.

Systems such as Alias Types [WM00] and Role Analysis [KLR02] specify the shape of a local object graph in more detail than our system. The Alias Types proposal uses this information to safely deallocate objects, while Role Analysis is used to specify and check properties of data structures. In contrast to these detailed specifications of a local alias graph, the goal of AliasJava is to provide a lightweight and practical way to constrain global aliasing within a program.

An alternative to using a type system to limit aliases is to use an alias analysis-based tool such as Lackwit [OJ97] to visualize the aliases within a program. For answering questions about aliasing, AliasJava can be more precise than Lackwit, which does not treat data structures polymorphically. Compared to Lackwit's successor Ajax [OCa00], AliasJava allows more parametric polymorphism on methods, but its treatment of subtype polymorphism is less precise due to the constraints of AliasJava's type system. One benefit of expressing alias information in a type system is that the information is constantly available and constantly checked for consistency, and so there is no need to run a tool to take advantage of it.

A final area of related work is systems that enforce the secure flow of information. A representative system is JFlow [Mye99], which annotates each piece of data with a set of principals that *own* the data, and for each owner, a list of principals that are allowed to *read* the data. The type system verifies that no principal can read a piece of data unless all the data's owners have give n read permission to that principal. AliasJava is more lightweight than JFlow, because our system labels references with a single owner instead of a list of owners and a list of authorized readers for each owner. However, our system only supports reasoning about information flow through data sharing, not other forms of information flow.

7. Conclusion

This paper described AliasJava, an annotation system for Java that places structural and temporal bounds on aliases, enabling developers to reason more directly about aliasing in object-oriented systems. AliasJava is expressive enough to describe a wide range of important idioms, including collection classes, iterators, and several architectural styles. Our design extends to the full Java language, including arrays, casts, inheritance, and inner classes. We formalized a subset of the system, and proved key invariants of the annotations. Our alias annotations can be automatically inferred using a novel variant of an existing instantiation constraint-based algorithm. We have validated the design of AliasJava and the inference algorithm on part of the Java standard library and on a realistic application. Our experience suggests that AliasJava is flexible enough to use on existing code, that annotation overhead is reasonable, and that the annotations can express important application constraints.

Acknowledgements

We would like to thank David Notkin, Doug Lea, members of the Cecil group, and the anonymous reviewers for their comments and suggestions. This work was supported in part by NSF grants CCR-9970986 and CCR-0073379, and gifts from Sun Microsystems and IBM.

References

[ACN02a] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. Proc. International Conference on Software Engineering, Orlando, Florida, May 2002.

[ACN02b] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning with ArchJava. Proc. European Conference on Object-Oriented Programming, Málaga, Spain, June 2002.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. University of Washington technical report UW-CSE-02-11-01, November 2002.

[Alm97] Paulo Sérgio Almeida. Balloon Types: Controlling Sharing of State in Data Types, Proc. European Conference on Object-Oriented Programming, Jyväskylä, Finland, June 1997.

[Arc02] ArchJava web site. http://www.archjava.org/

[Bok99] Boris Bokowski. Implementing "Object Ownership to Order." Proc. Intercontinental Workshop on Aliasing In Object-Oriented Systems, Lisbon, Portugal, June 1999.

[BN02] Anindya Banerjee and David A. Naumann. Representation Independence, Confinement, and Access Control. Proc. Principles of Programming Languages, Portland, Oregon, January 2002.

[BNR01] John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. Proc. European Conference on Object-Oriented Programming, Budapest, Hungary, June 2001.

[Boy01] John Boyland. Alias Burying: Unique Variables Without Destructive Reads. Software Practice & Experience, 6(31):533-553, May 2001.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. Proc. Object-Oriented Programming Systems, Languages and Applications, Tampa, Florida, October 2001.

[BR02] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. Proc. Object-Oriented Programming Systems, Languages and Applications, Seattle, Washington, November 2002.

[BS98] Boris Bokowski and André Spiegel. Barat—A Front-End for Java. Freie Universität Berlin Technical Report B-98-09, December 1998.

[Buc00] Alexander Buckley. Ownership Types Restrict Aliasing. MEng. Computing Final Year Project Report, Imperial College of Science, Technology and Medicine, London, United Kingdom, June 2000.

[BV99] Boris Bokowski and Jan Vitek. Confined Types. Proc. Object-Oriented Programming Systems, Languages, and Applications, Denver, Colorado, November 1999.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. Proc. International Conference on Software Engineering, Kyoto, Japan, April 1998.

[CD02] David Clarke and Sophia Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. Proc. Object-Oriented Programming Systems, Languages and Applications, Seattle, Washington, November 2002.

[Cla01] David Clarke. Object Ownership & Containment. Ph.D. Thesis, University of New South Wales, Australia, July 2001.

[CNP01] David G. Clarke, James Noble, and John M. Potter. Simple Ownership Types for Object Containment. Proc. European Conference on Object-Oriented Programming, Budapest, Hungary, June 2001.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. Proc. Object-Oriented Programming Systems, Languages and Applications, Vancouver, Canada, October 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed Memory Management in a Calculus of Capabilities. Proc. Principles of Programming Languages, San Antonio, Texas, January 1999.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. Proc. Programming Language Design and Implementation, Vancouver, Canada, June 2000.

[GHJ+94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison-Wesley, 1994.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-Based Memory Mangagement in Cyclone. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. Proc. Object-Oriented Programming Languages, Systems, and Applications, Tampa, Florida, November 2001.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Advances in Software Engineering and Knowledge Engineering, I (Ambriola V, Tortora G, Eds.) World Scientific Publishing Company, 1993.

[Hen93] Fritz Henglein. Type Inference with Polymorphic Recursion. Trans. Programming Languages and Systems, 15(2):253--289, April 1993.

[Hog91] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. Proc. Object-Oriented Programming: Systems, Languages and Applications, Phoenix, Arizona, October 1991.

[HLW+92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. OOPS Messenger, 3(2), April 1992.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. Proc. Object-Oriented Programming Systems, Languages, and Applications, Denver, Colorado, November 1999.

[KLR02] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role Analysis. Proc. Principles of Programming Languages, Portland, Oregon, January 2002.

[KTU93] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The Undecidability of the Semi-Unification Problem. Information and Computation, 102(1):83--101, January 1993.

[LHZ02] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using Data Groups to Specify and Check Side Effects. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

[Min96] Naftaly Minsky. Towards Alias-Free Pointers. Proc. of European Conference on Object Oriented Programming, Linz, Austria, July 1996.

[Mye99] Andrew C. Myers. JFlow: Practical Most-Static Information Flow Control. Proc. Principles of Programming Languages, San Antonio, Texas, January 1999.

[MP99] Peter Muller and Arnd Poetzsch-Heffter. Universes: A Type System for Controlling Representation Exposure. In A. Poetzsch-Heffter and J. Meyer (Hrsg.): Programmiersprachen und Grundlagen der Programmierung, 10. Kolloquium, Informatik Berichte 263, 1999/2000.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. Proc. European Conference on Object-Oriented Programming, Brussels, Belgium, 1998.

[OCa00] Robert O'Callahan. Generalized Aliasing as a Basis for Program Analysis Tools. Ph.D. Thesis, published as Carnegie Mellon technical report CMU-CS-01-124, November 2000.

[OJ97] Robert O' Callahan and Daniel Jackson. Lackwit:A Program Understanding Tool Based on Type Inference. Proc. International Conference on Software Engineering, Boston, Massachusetts, May 1997.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the Call-by-Value λ-Calculus Using a Stack of Regions. Proc. Principles of Programming Languages, Portland, Oregon, January 1994.

[Wad90] Philip Wadler. Linear Types Can Change the World! Programming Concepts and Methods, (M. Broy and C. Jones, eds.) North Holland, Amsterdam, April 1990.

[WM00] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. Proc. International Workshop on Types in Compilation, Montreal, Canada, September 2000.