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 dealing with 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. Data sharing is one of the main obstacles in understanding information flow within large software systems [HLW+92]. 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, any reference could point to any object of compatible type—and current languages do not provide much help in understanding them.

Data sharing problems can also compromise the security of a system. For example, in version 1.1 of the Java standard library, `Class.getSigners()`, one of the security system functions returned a pointer to an internal array, rather than a copy. This compromised the security of the "sandbox" that isolates Java applets, 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 allow software engineers to program in much the same style as in ordinary Java, but provide automatically checked documentation about data sharing within a program. We have also applied AliasJava as a way to specify the data sharing within a software architecture, as expressed in the architecture description language ArchJava [ACN02a].

There are two common forms of sharing in object-oriented systems. First, objects are shared in a structurally bounded way: an object is 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 statically grant trusted objects the capability to access its owned objects using a simple form of ownership parameterization. Second, objects are 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. For flexibility, 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 non-trivial programs as well as parts of the Java collection class library.

The rest of this paper is organized as follows. After the next section's discussion of related work, we introduce AliasJava in section 3. Section 4 formalizes our type system and outlines proofs of key properties. Section 5 describes our annotation inference algorithm. We evaluate our system in section 6 on several realistic systems and on the Java collection libraries, before concluding with a discussion of future work.

2. 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 combination allows us to express idioms that neither class of annotations can express alone (section 3.2).

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 type system [Boy01] described how to implement unique pointers without a special destructive read operation, an innovation adopted by AliasJava.

Linear type systems [Wad90] guarantee uniqueness and in addition can be used to track resource usage. Linear types have been applied to reason about protocols defining the order in which library methods can be called, as in the Vault language [FD02]. A number of research efforts have used linear types to reason about 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 make 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 for protecting access to the objects in a region; any object that can name the region can access the object inside it. AliasJava provides a capability-based encapsulation model, ensuring that owned objects can only be accessed by their owner and the objects to which the owner delegates access capabilities.

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. ESC/Java's specification system enables program-specific reasoning about *pivot objects* that are similar to our *owned* objects [DLN98]; however, their system could not guarantee that pivot objects are not accessed by unrelated parts of the program. 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.

The ownership annotations in AliasJava are most closely related to Flexible Alias Protection [NVP98] and its successors [CNP01,Cla01]. Flexible Alias Protection uses ownership polymorphism to strike a balance between guaranteeing aliasing properties and allowing flexible programming idioms. However, the original system did not support inheritance or common idioms like collection class iterators. Existing implementations of Flexible Alias Protection also lack support for language features such as inheritance [Bok99, Buc00], and thus there has been no significant experimental validation of this style of ownership types.

Clarke *et al.* extend Flexible Alias Protection to handle iterators and inheritance by creating the notion of an object's *representation context*, which is separate from the object [CNP01]. 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. Thus, the representation context of an iterator can be defined to be within the representation of the collection it iterates over, allowing the iterator to refer to the collection's implementation.

Rather than enforcing an invariant over owning and representation contexts, AliasJava introduces a capability-based model for reasoning about ownership. Collection classes can delegate a capability to their iterators, and use object-oriented subsumption to hide that capability from external classes, as described in section 3.2.1. We extend Flexible Alias Protection to the full Java language, handling issues such as casting that are not adequately considered in the previous literature. This paper also provides the first significant experimental validation of parameterized ownership types. We are also the first to define a global annotation inference algorithm for ownership types. Finally, we extend Flexible Alias Protection with *lent* and *unique* references and formalize the system in a subset of Java. Comparable earlier formalizations either neglected inheritance [CPN98] or formalized ownership types in a more abstract object calculus [Cla01].

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 low-level 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]. Although it provides powerful concurrency guarantees, PRFJ does not support encapsulation: if a program can access an object, it can also access that object's representation. AliasJava supports a strong capability-based encapsulation model, in which an object can only access a second object's representation if the latter object has statically delegated a capability to the former object.

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 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

```
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

polymorphism is less precise due to the constraints of AliasJava's type system.

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 given 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.

3. 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 to describe the extent of sharing of that reference. The annotations bound aliasing structurally: *unique* describes an unshared reference, *owned* objects are assigned an *owner* that controls who may access that object, and *shared* indicates the worst case of a globally-aliased reference. We also provide a *lent* annotation for 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, providing global guarantees about aliasing properties. However, it may require a significant effort to annotate a large legacy program with alias annotations. Therefore, 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 6. A promising alternative is inferring alias annotations for a closed subset of the program automatically. Section 5 presents an annotation inference algorithm, and we present early results from a prototype implementation.

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

Subsection 3.1 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 4). Subsection 3.2 shows more examples of the language in order to illustrate its expressiveness. We discuss the reasoning benefits provided by our annotation system in subsection 3.3.

3.1. Annotations for Data Sharing

In this subsection, we first describe the various annotations in AliasJava through a series of examples. Next, we describe the properties guaranteed by our annotations. Finally, we discuss how we handle the features of the full Java language.

3.1.1. Annotation System Design

Design principles. 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.

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. Unique annotations can also aid software evolution by documenting the absence of aliases to an object. Figure 1 illustrates uniqueness through a linked list class where all of the elements and all of the links are **`unique`**.

In general, when a **`unique`** variable or field is read, the source location must be dead—otherwise the read reference would be an

```
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();
  }
}

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);
  }
}

class Link<element, link> {
  public Link(element Object obj,
              link Link<element, link> next) {
    this.obj = obj; this.next = next;
  }
  public element Object member() {
    return obj;
  }
  public link Link<element, link> next() {
    return next;
  }
  link Link<element, link> next;
  element Object obj;
}
```

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.

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.

In AliasJava, **`unique`** can be considered a universal capability: **`unique`** values can be assigned to a location with any other data sharing annotation, representing a weaker capability. The converse is not true, as the other data sharing annotations do not guarantee that a value is unique.

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. Also, if the rectangle is

```
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

being displayed on a screen, it must be made aware of any changes to its shape so that it can update the screen.

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, 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 cannot escape the scope of the enclosing object, enabling reasoning about object containment. This allows the implementer of `Rectangle` to rely on the fact that external objects can neither change nor see changes to its representation, except through the object'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. Many container classes have their own internal representation, but are used by clients to store external objects. In this case, we can pass the owner of the elements as a 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 6.1.

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 of the stack.

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.

```
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

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, they are essential for interoperating with existing run-time libraries, legacy code, and static fields, which may refer to aliases that are not confined to the scope of any object instance.

Lent. Figure 5 shows a `contains` method that checks if an integer is stored in a linked list created using the `LinkedList` class from Figure 1, which consists of unique elements and links. 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 duplicating 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 are guaranteed to be dead 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, **`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`** is a weak capability in that **`lent`** values may only be assigned to other **`lent`** locations.

Summary. The table below 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 (unique references can be assigned to a location of any type annotation), and that **`lent`** is a universal sink (**`lent`** locations can be assigned a value with any type annotation). The other type annotations must be kept separate from each other.

	`unique`	**`owned`**	**`shared`**	**`lent`**
`unique`	X	X	X	X
`owned`		X		X
`shared`			X	X
`lent`				X

3.1.2. Properties

AliasJava ensures uniqueness and ownership invariants that restrict the aliasing patterns that can occur during program execution. Section 4 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 annotated with an owner `o'` refers to an 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 are annotated with an owner `o'`.

Another way to state the ownership invariant is that each non-**`unique`**, non-**`shared`** object is owned by exactly one other object. Only an object's owner, and the objects that the owner has delegated a capability to, may store a reference to that object. 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. In contrast, answering this question requires a sophisticated alias analysis in systems like PRFJ, where merely having a reference to an object gives a program permission to access that object's representation [BR01].

3.1.3. Java Integration

The Java language has a number of 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 declare the alias parameters of the classes and interfaces it extends and implements. 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 and constructors, 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. In the case of constructors, the annotation of **this** determines the type annotation of the value returned from **new** expressions that invoke that constructor. Because the vast majority of methods and constructors have a **lent** annotation, this is the default in our system and need not be explicitly specified.

Inner Classes. Inner classes implicitly import the parameters α of their surrounding class. Their qualified type 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 type **owned**. The inner class can have its own additional parameters, if necessary. Inner classes constructed 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 thus they follow the same rules as ordinary classes.

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 (of course, 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.

Arrays. An array must be declared with an alias type for each of the levels in the array. For example, the type `List<•>[owned]` refers to an **owned** array of lists that hold objects of alias type •.

Concurrency. Concurrency is largely orthogonal to this work. However, in order to preserve uniqueness, 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), and only if the field is set to another value before the end of the synchronization block. To support the stronger invariant that only one thread at a time may access a **unique** object, a compiler implementation may produce a warning even for reads of a **unique** field into a **lent** location that are not wrapped by synchronization.

Casts. Because a class may extend a class with fewer parameters, alias parameters may be hidden by subsumption. In order to recover this parameterization information safely, each parameterized class must store 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, which are likely to be heavyweight objects in any case. 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

```
interface Iterator<elem_owner> {
  elem_owner 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

object corresponds to the formal parameter; and 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 also needs to be passed to methods that have alias parameters.

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 subsumption and casts in a way that does not violate the semantics of the type annotations.

The Java Standard Library. We have chosen to implement our system on top of the standard JVM, which meant that we could not modify the bytecode of the Java standard library. We can still add source-level alias annotations to the standard library, and check client code against the library. However, Java's reflection reflection interfaces provide a way to get around the 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 the standard library cannot be modified, we cannot 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. For example, one could distribute a JVM with custom library code that does generate the run-time information, or check for consistency between multiple casts on the same object, or even conservatively reject all casts that do not have the required run-time alias parameter information.

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

```
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 as a `unique` reference from a client method.

3.2. Examples

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

3.2.1. 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.

3.2.2. 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. Detlefs *et al.* describe the `Lexer` class in Figure 7, which accepts an input stream that becomes part of its representation [DLN98]. 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 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. To the best of our knowledge, no previous system can express this idiom while providing an initial guarantee of uniqueness and later guaranteeing that the stream is encapsulated within the `Lexer`.

3.2.3. 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].

```
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);
 }
}

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.

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.

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

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 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 data store accepts an alias parameter describing the owner of the data that it stores. In its `data` port, the data store defines a **`requires`** method that it calls to notify clients whenever data has changed. It also implements two methods allowing clients to get data and to update the 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. In addition, the signatures of the methods in these components' ports show which data structures may be shared. For example, change messages are owned by the data store and are only temporarily passed to the computational components, and the reverse invariant is true for the specifications of what data to get. The actual data, however, is annotated with the `data_owner` parameter, indicating that it is shared persistently between different components in the architecture.

```
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);
  }
}

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;
}
```

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

3.3. 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.

Although AliasJava contains some imprecision (particularly for **`shared`** references), it has the advantage over alias visualization tools that the data sharing information is explicit in the source code—there is no waiting for an alias analysis to run, and no need even to context-switch away from editing the code to running a tool. Thus maintainers may benefit from the system when they would not otherwise take the initiative to consult an outside tool.

What components might this component communicate with? It is important to answer this question when making changes to a large software system. ArchJava makes control flow between components explicit in the connections between components. The examples in Figures 8 and 9 show that communication through shared data is made explicit through alias annotations, complementing the existing ArchJava language.

How difficult would it be to distribute the architecture 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 can help programmers to anticipate the issues likely to come up when distributing a program across multiple machines. For example, objects annotated **`lent`** or **`unique`** can probably be passed by value between two distributed components. On the other hand, if the alias annotations in the architecture indicate sharing between components, either a solution using remote object references or extensive refactoring of the source code will be necessary.

```
CL      ::= class C<α,β> extends D<α> {T f; M}

M       ::= T m(T x) T { return e; }

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

v       ::= null
        |   ℓ

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

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

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

ℓ       ∈ Locations
α,β     ∈ Parameters
```

Figure 10. AliasFJ Syntax

4. 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).

4.1. AliasFJ

Syntax. Figure 10 presents the syntax of AliasFJ. The metavariables C, D and E range over class names; 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.

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 $S[\ell]$ to denote the store entry for ℓ, and $S[\ell, i]$ to denote the value in the *i*th field of $S[\ell]$. Functional store updates are abbreviated $S[\ell \rightarrow C<\overline{\ell}>(\overline{v})]$. The store also holds the actual alias parameters for each location, in order to check run-time casts properly.

$$\frac{}{C <: C} \quad \text{(CLASS-REFLEX)}$$

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

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

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

$$\frac{}{\textbf{ERROR} <: T} \quad \text{(SUBTYPE-ERROR)}$$

$$\frac{}{\textbf{NULL} <: T} \quad \text{(SUBTYPE-NULL)}$$

Figure 11. Subtyping Rules

Classes define a set of fields $\overline{f}$ and methods $\overline{M}$. Expressions include primitive values, 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. 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, and thus we include locations in the type syntax so that we can give alias types to expressions in an executing program.

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

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

$$\frac{\begin{array}{c} S[\ell] = C<\bar{\ell}>(\bar{v}) \qquad \mathit{fields}(C) = \bar{T}\ \bar{f} \\ T_i = A_i\ D_i<\bar{\beta}> \\ A_R = [\texttt{lent/unique}]\,[\ell/\texttt{owned}]\ A_i \end{array}}{S \vdash A(\ell).f_i \rightarrow A_R(v_i),\ S} \qquad \text{(R-READ)}$$

$$\frac{\begin{array}{c} S[\ell] = C<\bar{\ell}>(\bar{v}) \qquad \mathit{fields}(C) = \bar{T}\ \bar{f} \\ T_i = \texttt{unique}\ C<\bar{\alpha}> \\ S' = S[\ell \rightarrow C<\bar{\ell}>([\texttt{null}/v_i]\bar{v})] \end{array}}{S \vdash \texttt{unique}(A(\ell).f_i) \rightarrow \texttt{unique}(v_i),\ S'} \qquad \text{(R-UNIQUEREAD)}$$

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

$$\frac{S[\ell] = D<\bar{\ell}>(\bar{v}) \qquad A\ D<\bar{\ell}> <: T_c}{S \vdash (T_c)A(\ell) \rightarrow A(\ell),\ S} \qquad \text{(R-CAST)}$$

$$\frac{\begin{array}{c} S[\ell] = C<\bar{\ell}>(\overline{v_f}) \qquad CT(C) = \texttt{class}\ C<\bar{\alpha}>\ldots \\ \mathit{mbody}(m, C) = (\bar{x}, e) \qquad \overline{v_{\texttt{lent}}} = [\texttt{lent/unique}]\ \bar{v} \\ \texttt{this}_{\texttt{lent}} = [\texttt{lent/unique}]\ A_0(\ell) \\ e' = [\bar{v}/\texttt{unique}(\bar{x}),\ A_0(\ell)/\texttt{unique}(\texttt{this})]\,e \\ e'' = [\bar{\ell}/\bar{\alpha},\ \overline{v_{\texttt{lent}}}/\bar{x},\ \texttt{this}_{\texttt{lent}}/\texttt{this}]\,e' \end{array}}{S \vdash A_0(\ell).m(\bar{v}) \rightarrow e'',\ S} \qquad \text{(R-INVK)}$$

Figure 12. AliasFJ Evaluation Rules

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 given in Figure 12, is of the form $S \vdash e \rightarrow e', S'$ read "In the context of store S, expression e reduces to expression e' in one step, producing the new store S'." We write $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$. 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 updated 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 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.

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 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 15) 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 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**.

The full semantics of the language include error rules representing casts that fail and null pointer dereferences. A set of congruence rules (such as if $e \rightarrow e'$ then $e.f \rightarrow e'.f$) allows reduction to proceed in the order of evaluation defined by Java, and similar rules are defined to propagate error subexpressions outward. We omit the congruence and error rules here.

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

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

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

$$\frac{\begin{array}{c}\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{A}\ \mathtt{C}\langle\overline{\alpha}\rangle \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \mathtt{T}_i = \mathbf{unique}\ \mathtt{D}\langle\overline{\beta}\rangle \qquad \mathtt{T}_R = \mathit{inst}(\mathtt{T}_i, \mathtt{A}\ \mathtt{C}\langle\overline{\alpha}\rangle)\end{array}}{\Gamma, \Sigma \vdash \mathbf{unique}(\mathtt{e}.\mathtt{f}_i) : \mathtt{T}_R} \quad \text{(T-UFIELD)}$$

$$\Gamma, \Sigma \vdash \mathbf{new}\ \mathtt{C}\langle\overline{p}\rangle(\overline{\mathtt{e}}) : \mathbf{unique}\ \mathtt{C}\langle\overline{p}\rangle \quad \text{(T-NEW)}$$

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

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

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

$$\frac{\Sigma(\ell) = \mathtt{C}\langle\overline{\ell}\rangle}{\Gamma, \Sigma \vdash \mathtt{A}(\ell) : \mathtt{A}\ \mathtt{C}\langle\overline{\ell}\rangle} \quad \text{(T-LOC)}$$

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

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

Figure 13. AliasFJ Typechecking

Typing Rules. Typing judgments, shown in Figure 13, 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 15), 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.

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

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{T}_{\mathtt{this}}, \varnothing \vdash \mathtt{e} : \mathtt{S} \qquad \mathtt{S} <: \mathtt{T} \\ \mathit{CT}(\mathtt{C}) = \mathbf{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\ \mathbf{extends}\ \mathtt{D}\ldots \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \mathtt{T}_{\mathtt{this}} \times \overline{\mathtt{T}} \to \mathtt{T}) \qquad \mathtt{T}_{\mathtt{this}} = \mathtt{A}\ \mathtt{C}\langle\overline{\alpha}\rangle \\ \forall \mathtt{x} \in \{\overline{\mathtt{x}}, \mathtt{this}\}\ .\ \mathbf{unique}(\mathtt{x})\ \text{appears at most once in}\ \mathtt{e}\end{array}}{\mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \mathtt{T}_{\mathtt{this}}\ \{\ \mathbf{return}\ \mathtt{e};\}\ \mathtt{OK\ in\ C}} \quad \text{(T-METH)}$$

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

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

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

Figure 14. AliasFJ Class, Method, and Store Typing

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 "stupid 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 14 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 "class 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 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 clauses of the store typing rule are the usual well-formedness rules, requiring the store type Σ to type every location in S, and verifying that the types of objects in a field are compatible with the field's type. The last rule defines the *annotation* convenience function, which is used in stating the properties of the alias annotation system.

Field lookup:

$$fields(\texttt{Object}) = \bullet$$

$$\frac{CT(\texttt{C}) = \texttt{class C}\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends D}\langle\overline{\alpha}\rangle\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad fields(\texttt{D}) = \overline{\texttt{T}_g}\ \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{T}_g}\ \texttt{g},\ \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

Method type lookup:

$$\frac{CT(\texttt{C}) = \texttt{class C}\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends D}\langle\overline{\alpha}\rangle\ \{\overline{\texttt{T}_F}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \texttt{T m}\ (\overline{\texttt{T}}\ \overline{\texttt{x}})\ \texttt{T}_{\texttt{this}}\ \{\ \texttt{return e;}\ \} \in \overline{\texttt{M}}}{mtype(\texttt{m},\texttt{C}) = \texttt{T}_{\texttt{this}} \times \overline{\texttt{T}} \to \texttt{T}}$$

$$\frac{CT(\texttt{C}) = \texttt{class C}\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends D}\langle\overline{\alpha}\rangle\ \{\overline{\texttt{T}_F}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \texttt{m}\ \text{is not defined in}\ \overline{\texttt{M}}}{mtype(\texttt{m},\texttt{C}) = mtype(\texttt{m},\texttt{D})}$$

Method body lookup:

$$\frac{CT(\texttt{C}) = \texttt{class C}\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends D}\langle\overline{\alpha}\rangle\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \texttt{T m}\ (\overline{\texttt{T}_x}\ \overline{\texttt{x}})\ \texttt{T}_{\texttt{this}}\ \{\ \texttt{return e;}\ \} \in \overline{\texttt{M}}}{mbody(\texttt{m},\texttt{C}) = (\overline{\texttt{x}},\texttt{e})}$$

$$\frac{CT(\texttt{C}) = \texttt{class C}\langle\overline{\alpha},\overline{\beta}\rangle\ \texttt{extends D}\langle\overline{\alpha}\rangle\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\} \quad \texttt{m}\ \text{is not defined in}\ \overline{\texttt{M}}}{mbody(\texttt{m},\texttt{C}) = mbody(\texttt{m},\texttt{D})}$$

Alias type instantiation:

$$\frac{CT(\texttt{D}) = \texttt{class D}\langle\overline{p}\rangle\ldots}{inst(\texttt{T},\texttt{B D}\langle\overline{q}\rangle) = [\overline{q}/\overline{p}]\texttt{T}}$$

Valid method overriding:

$$\frac{mtype(\texttt{m, C}) = \texttt{T}_{\texttt{this}} \times \overline{\texttt{T}} \to \texttt{T}_0 \Rightarrow \quad \overline{\texttt{U}} = \overline{\texttt{T}} \ \wedge\ \texttt{U}_0 = \texttt{T}_0 \ \wedge\ \texttt{U}_{\texttt{this}} = \texttt{A C}_\texttt{U}\langle\overline{\alpha},\overline{\beta}\rangle \ \wedge\ \texttt{T}_{\texttt{this}} = \texttt{A C}_\texttt{T}\langle\overline{\alpha}\rangle}{override(\texttt{m, C, U}_{\texttt{this}} \times \overline{\texttt{U}} \to \texttt{U}_0)}$$

Figure 15. AliasFJ Auxiliary Definitions

Auxiliary Definitions. Most of the auxiliary definitions shown in Figure 15 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.

4.2. Type Soundness

We prove 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 other than failed casts and null-pointer exceptions.

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

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 $\overline{\texttt{x}}:\overline{\texttt{T}}$, $\texttt{this}:\texttt{T}_{\texttt{this}}$, $\varnothing \vdash \texttt{e}:\texttt{T}$, $\varnothing,\texttt{S} \vdash \overline{\texttt{v}}:\overline{\texttt{S}}$, $\varnothing,\texttt{S} \vdash \texttt{A}(\ell):\texttt{T}_\texttt{A}$, $\overline{\texttt{S}} <: [\ell/\alpha, \ell/\textbf{owned}]\overline{\texttt{T}}$, $\texttt{T}_\texttt{A} <: [\ell/\alpha, \ell/\textbf{owned}]\texttt{T}_{\texttt{this}}$ and $\Gamma,\Sigma \vdash \texttt{S}$, then $\varnothing, \texttt{S} \vdash [\overline{\texttt{v}}/\overline{\texttt{x}}, \ell/\texttt{this}, \ell/\alpha, \ell/\textbf{owned}]\texttt{e}:\texttt{T}'$ for some $\texttt{T}' <: [\ell/\alpha, \ell/\textbf{owned}]\texttt{T}$.

The proof is by induction over the structure of `e`, with a case analysis on the form of the outermost term.

Subject reduction is then proved by induction on the derivation of $\texttt{S} \vdash \texttt{e} \to \texttt{e}',\texttt{S}'$, with a case analysis on the last reduction rule used. □

Theorem [Progress]: If $\varnothing,\Sigma \vdash \texttt{e}:\texttt{T}$, then either `e` is an irreducible value, an **`error`**, or else $\forall\texttt{S}$ such that $\varnothing,\Sigma \vdash \texttt{S}$, $\texttt{S} \vdash \texttt{e} \to \texttt{e}',\texttt{S}'$.

The proof is by induction on the derivation of $\varnothing,\Sigma \vdash \texttt{e} \in \texttt{T}$, with a case analysis on the last typing rule used. □

4.3. 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 references refer to that location.

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

Formally, we say that ℓ occurs in `S` with annotation `A` if there exists some ℓ', i such that $\texttt{S}[\ell'][i] = \ell$ and $annotation(\texttt{S},\ell',i)=\texttt{A}$. We say that ℓ occurs in `e` with annotation `A` if $\texttt{A}(\ell)$ 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 proof is by induction on the derivation of $\varnothing \vdash \texttt{e} \to^* \texttt{e}',\texttt{S}'$, with a case analysis on the last reduction rule used. 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 most interesting cases in the proof are method invocation, where the method typing rule ensures that unique arguments are not duplicated during method substitution, and unique field read, where the semantics of the rule assigns **`null`** to the read field. □

We have argued in section 3.1.2 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 $\varnothing,\varnothing \vdash e:T$ and $\varnothing \vdash e \rightarrow^* e',S'$, 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 $\varnothing \vdash e \rightarrow^* e',S'$, 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. □

Corollary [Ownership Soundness]: If $\varnothing,\varnothing \vdash e:T$, $\varnothing \vdash e \rightarrow^* e',S'$ 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. □

5. 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 which depends only on the inferred **lent** annotations. Finally, we infer the remaining annotations in a final pass.

5.1. 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. 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 showing 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**.

5.2. 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 killing assignments ($x =_{kill} y$), where y is dead after the assignment. We assume that live variable analysis has already annotated all value flows as ordinary assignment or killing 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 killing assignments ($x =_{kill} y$) is simple: if y is **non-unique**, than x must be **non-unique** also. The intuition here is that 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 killing 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.

5.3. 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 [Hen83]. 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 a 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 by the 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.

Due to space constraints, we cannot present the full details of the inference algorithm, which will be described in a forthcoming technical report. Instead, we present a high-level overview of the

algorithm in parallel with an example that illustrates many of the key issues. We choose as our running example the `Stack` code in Figure 3, assuming initially that none of the alias annotations in that figure are 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 source expression. Distinct type variables indicate distinct alias parameters of the enclosing class.

Figure 16 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. We have also eagerly merged variables like the three "`next`s" that would eventually be merged in any case.

Our analysis solves three different forms of constraints: equality, component, and instantiation, which are described in turn below.

Equality Constraints. When value flows from one variable to another within a class, we generate an equality constraint ($a = b$), indicating that two alias 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 unify the corresponding *pop* and *member* variables, because that would place unnecessary constraints on other parts of the program that use `List.member`. We use instantiation constraints (discussed below) to reason about value flow between classes in a way that treats different `List` objects differently.

In our implementation, equality constraints are solved 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. Figure 16 shows the equality constraints generated from Figure 3.

The initial equality constraints shown at the top of Figure 16 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 containment and instantiation constraints.

Component Constraints. A component constraint ($o \triangleright_f v$), read "*v* is a component of *o* with index *f*," means that the type variable *v* represents the *f* field 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

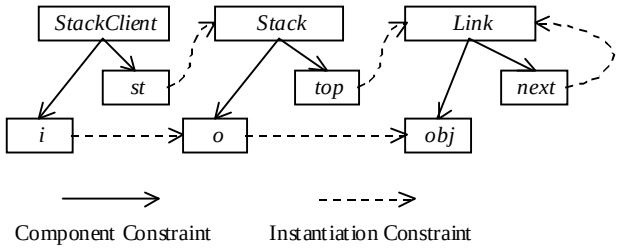
Initial variables:

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

Initial equality constraints:

top = temp
member = obj

After solving initial equality & uniqueness constraints:



Final constraint system:

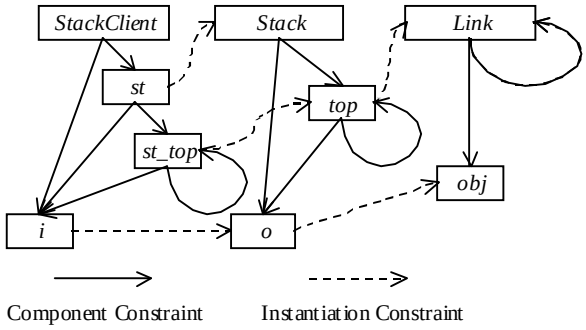


Figure 16. Constraints generated and solved during inference of the alias types given in Figure 3.

field `f` of a class `C`, we generate a component constraint $C \triangleright_f f$. We generalize the notion of fields to any type variable within a class, so that component constraints are also generated for method arguments, results, and local variables.

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

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` field of the list 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* list. In both this case

and the case above, the index on the instantiation constraint shows how the instance is related to its parent.

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*. 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:

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

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 entirely analogous rule is used to ensure that two components with the same index are also unified:

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

The first diagram in Figure 16 shows the example system after solving the initial equality and uniqueness constraints.

Constraint Propagation. If `top` is an instance of `List`, as shown by the constraints in Figure 16, 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 `List`. This motivates the component propagation rule:

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

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

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 copy of *next* that is a component of the *top* instance of *List*. We encode this intuition with the constraint that *top_next* is an instance of *next*. Then *top_next* will be a transitive instance of *List*, ensuring that it will gain its own *next* and *obj* components. These constraints are generated with the instance propagation rule:

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

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. The rule's conclusion, in the case of *top_next*, simply states that *next* $\leq_{top}$ *top_next*.

Avoiding infinite propagation. The discussion above suggests that constraint propagation as presented above may never terminate. For example, *top* is a *List*, so it must have a *next* component *top_next*. But, *top_next* is transitively a *List* 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 regress if we are 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:

$$\text{If } \exists\, L \leq_{i1} a_1 \leq_{i2} \ldots \leq_{iN} R \text{ and } \exists\, L \rhd_{c1} b_1 \rhd_{c2} \ldots \rhd_{cM} R$$
$$\text{then } L = R$$

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 *List* $\rhd_{next}$ *next* $\wedge$ List $\leq_{next}$ *next*. Thus, our implementation generates the equality constraint *next=List*, which eliminates the source of the loop.

The diagram at the bottom of Figure 16 shows the final results of the constraint-based algorithm. As described above, *next* has been unified into *List*. 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 *List* 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 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.

Precision. Application of the extended occurs check rule may result in imprecise types, because it imposes unification that is not required for correctness. For example, we could be more precise if we unified *List* with *next_next*, instead of *next*, enabling us to encode lists with alternating alias types. We expect that additional expressiveness of this type is not typically useful, and in fact our implementation works well in practice despite the small loss of precision due to this eager unification.

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` 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 for each 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. Since owned nodes are by definition inaccessible from outside an object, owned nodes that are components of another node need not be propagated using the propagation rule. This optimization, first identified by

O'Callahan [OCa00], has the potential to save a significant amount of space and time in the analysis.

The **shared** flag is initialized to **true** for each **static** field and each argument and result of a **static** method, as these are the base cases for **shared** nodes. 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. When a new type variable is created due to the component propagation rule, the **lent** and **unique** flags of the source component are simply copied to the newly created component.

Lent and unique annotations must be treated specially at merges, because a **unique** value could flow to two nodes with different alias types (presumably along different program paths). Likewise, values with two different alias annotations could legally flow to the same **lent** node. In both cases, the alias annotations of nodes that flow to and from **lent** and **unique** may differ, but the actual alias parameters of these nodes (represented by their component constraints) must be the same. We implement these semantics by separating the type variables used to represent a node's alias annotation from the type variable used to reason about that node's components. Equality constraints between non-lent, non-unique nodes merge both type variables, implementing the (simplified) semantics discussed above. However, when at least one of the nodes in an equality constraint is **lent** or **unique**, our algorithm unifies only the type variables representing the alias parameters of the node.

Final alias annotations. The final alias annotations are assigned from the constraint graph so as to make the annotations as precise and general as possible. Since **lent** is the most general annotation, all type variables whose node has a lent flag equal to true are given a **lent** annotation. Unique is the most precise annotation for the remaining nodes, so any remaining node with a true unique flag is annotated **unique**. In order to be sound, we must next make any unmarked nodes with a true shared flag **shared**. Next, we mark the remaining nodes as **owned** based on their owned flags. Any remaining nodes are parameters; for each class, the different type variables that are components of that class are given letter names `a`, `b`, `c`, and so forth. Nodes that have components are given actual alias parameters based on the component type: either **owned**, **shared**, or a formal alias parameter of the enclosing class.

6. 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 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.

6.1. Hashtable

Motivation. Collection class code is a challenge for alias annotation systems, because collection classes 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.

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

- Can the annotation system effectively express the aliasing constraints of collection class code?
- How much effort does the annotation system take?
- 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 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 annotating a local part of a system can be effective.

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 be expensive 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 then `entry` and `key` parameters. On the other hand, `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>`, at the cost of making the hash table harder to understand and use because of its six alias parameters. 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 `nextElement` to always return an entry, and then defined two wrapper classes that implement `Iterator` and extract and return the key and value from the `Hashtable` 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 boolean contains(lent Object o) {
      return containsKey(o);
    }
    // other methods...
  }
```

In this code, class `Keyset` can reference all of the parameters of the enclosing `Hashtable`, simplifying its alias annotations considerably.

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 could 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 `ObjectInputStream` and `ObjectOutputStream`. Our annotation system expressed the conceptual semantics of these serialization-related classes (e.g., `writeObject` accepts a **`lent`** argument and `readObject` returns a **`unique`** object). However, the actual implementation of these classes must cache object references in order to reconstruct 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 correct 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).

6.2. 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 [ACN01a]. 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 parameters 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.

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 shared objects passed between components came from a small set of classes including circuit elements and data structures that reflect their organization into a circuit.

Future Work. Despite the soundness of AliasJava, the annotations we added to Aphyds' architecture may be inaccurate if the annotations we asserted for other parts of the application and standard library were erroneous. In future work, we intend to apply our annotation inference algorithm to address this issue by either automatically generating or automatically checking these assertions.

6.3. Annotation Inference

We evaluate our annotation inference algorithm in several ways. First, we apply inference to small examples, and compare the inferred types with those generated by hand. Second, we collect statistics regarding **`lent`** and **`unique`** inference on the Java standard library, to determine how often our algorithm is able to infer these optimistic types.

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. Annotation inference also generated an extra, unused parameter for the `Iterator` interface in Figure 6; this appears to be a defect in our implementation.

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.util.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 constraint solver implementation is still an early prototype, and has not been optimized for execution time or space consumption. The solver can infer alias parameters for the 408 Java standard library classes in about 12 minutes, using 1.5 GB of memory. O'Callahan discusses a number of optimizations that can reduce space and time of instantiation constraint-based inference systems [OCa00]; we hope to improve our algorithm's efficiency by applying these optimizations.

Since our inference algorithm has a compositional variant, it can be applied to one strongly-connected component of the program at a time, rather than to the full program. Thus, even if our final, optimized inference algorithm scales only to a certain size, it is still applicable to any program that can be broken down into strongly-connected components of no greater than that size.

Standard Library Statistics. We collected statistics about inferred **`lent`** and **`unique`** types on the standard library. We ran the analysis on the Java standard library from the JDK 1.2.1, inferring types for all classes statically reachable from `java.lang.Object`. A total of 408 classes were traversed, including 932 constructors and 3682 instance methods.

	% lent	% unique
Method parameters	34-46	16
Constructor parameters	22-25	31
Method **this**	68-77	0
Constructor **this**	76-95	0
Method return type	N/A	25-31
Fields	N/A	18-22

The table above shows the percentage of lent and unique annotations on various different syntactic constructs. We report a range of percentages for each value, with the higher number comes from making optimistic (**lent** arguments and **unique** results) assumptions about native method signatures, and the lower number is from conservative assumptions about native methods. The figures shown were collected using our non-compositional inference algorithm. The compositional version yields essentially the same results, except that it infers many fewer **unique** method and constructor parameters, because only **private** methods may have **unique** arguments.

The table shows that our annotation inference algorithm was able to effectively capture common object-oriented programming styles. For example:

- Most constructors have **lent** annotations for **this**, meaning that **new** will return a **unique** pointer.
- Most methods don't store the **this** parameter in any field, so it is annotated **lent**. Therefore, these methods are usable on **unique** objects without giving up the uniqueness.
- Many methods use their parameters only for the length of the method call.
- Constructors have fewer **lent** parameters because they often store their parameters into fields of the constructed object. However, they often have **unique** parameters because clients often create a constructor's arguments in the **new** expression.
- A significant number of methods are factory methods, returning a **unique** reference.

We will include a more complete analysis in our final paper, but our early results suggest that annotation inference can describe interesting aliasing properties in the Java standard library.

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. We have described our handling of many of Java's language features, and formalized the system in a subset of Java, proving key invariants of the annotations. Our algorithm for inferring alias annotations is the first to apply to ownership types, using a novel variant of existing instantiation constraint systems. Finally, we have validated the design of AliasJava and the inference algorithm on parts 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, 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, FL, 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.

[Alm97] P.S. Almeida. Balloon Types: Controlling Sharing of State in Data Types, Proceedings of ECOOP '97, pp. 32-59, 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, June 1999.

[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. In Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Tampa Bay, FL, Oct. 2001.

[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. In Proceedings 14th Annual ACM SIGPLAN Conference on ObjectOriented Programming Systems, Languages, and Applications, Denver, Colorado, USA, November 1999.

[CBS98] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, April 1998.

[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. In European Conference for Object-Oriented Programming, June 2001.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for Flexible Alias Protection. In OOPSLA'98 Conference Proceedings---Object-Oriented Programming Systems, Languages and Applications, Vancouver, Canada, October 18--22, ACM SIGPLAN Notices, 33(10):48--64, October 1998.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In Alex Aiken, editor, Proc. 26th Annual ACM Symposium on Principles of Programming Languages, pages 262--275, San Antonio, Texas, USA, January 1999. ACM Press.

[DLN98] David Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Compaq Systems Research Center, Palo Alto, CA, July 1998.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. To appear in Proceedings of the ACM Conference on Programming Language Design and Implementation, June 2002.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In Conference on Programming Language Design and Implementation, pages 253--263, 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. In Conference on Programming Language Design and Implementation, June 2002.

[GPV01] Grothoff, Palsberg, and Vitek, Encapsulating Objects with Confined Types, in Proceedings of ACM SIGPLAN 2001 Conference on Object-Oriented Programming Languages, Systems, and Applications, 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. ACM Transactions on Programming Languages and Systems, 15(2):253--289, April 1993.

[Hog91] J. Hogg. Islands: Aliasing protection in object-oriented languages. In Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91), volume 26, pages 271--285. ACM SIGPLAN Notices, 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 Igarishi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Proceedings of ACM Conference on Object Oriented Languages and Systems, November 1999.

[KLR02] Viktor Kuncak, Patrick Lam, and Martin Rinard: Role Analysis. Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages. Portland, OR, January 2002.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. Information and Computation, 102(1):83--101, Jan. 1993.

[LV95] D.C. Luckham, J. Vera. An Event Based Architecture Definition Language. IEEE Transactions on Software Engineering Vol. 21, No 9, September 1995.

[Min96] Naftaly Minsky. Towards Alias-Free Pointers. Proc. of the 10th European Conference on Object Oriented Programming (ECOOP96), Linz, Austria July 1996.

[Mye99] Andrew C. Myers. JFlow: Practical Most-Static Information Flow Control. In Proceedings of the Symposium on 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. 12th 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, MA, May 1997.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, 17:40--52, October 1992.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value λ-calculus using a stack of regions. In Proceedings of the 21st ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, pages 188-- 201. ACM Press, 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. In International Workshop on Types in Compilation, Montreal, Canada, September 2000.