JavaD: Bringing Ownership Domains to Mainstream Java

Marwan Abi-Antoun Jonathan Aldrich

May 2006
CMU-ISRI-06-110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Unlike many proposed designs for ownership type systems, AliasJava has had a publicly available implementation for a few years and has been applied on several case studies. However, AliasJava is currently implemented as a non-backwards compatible extension of Java. As a result, none of the tool support for Java programs is available for AliasJava programs, making it harder to justify the case that Java programs are easier to evolve with AliasJava annotations than without. Furthermore, using language extensions makes it harder to specify the ownership and aliasing annotations for a large legacy system since the program cannot be annotated partially and incrementally with AliasJava.

We present and evaluate JavaD, a re-implementation of the AliasJava language and analysis as a set of Java 1.5 annotations, using the Eclipse Java Development Tooling (JDT) infrastructure and the Crystal Data Flow Analysis framework. We conclude with limitations, lessons learned and future plans.

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, a 2004 IBM Eclipse Innovation Grant, the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

Keywords: ownership domains, AliasJava, type annotation

1 Introduction

"The big lie of object-oriented programming is that objects provide encapsulation" [13]. In particular, aliasing can cause a failure of encapsulation and can be the source of many unintended side effects in object-oriented programs (See Figure 1).

Aliasing cannot be eliminated entirely from useful object-oriented program; in fact, many object-oriented design patterns rely on it. However, the research community has recognized that aliasing must be controlled using language support (see [8] for a comprehensive survey) and proposed several solutions, e.g., Islands [13] and Universes [15] to name only a few.

We look in particular at one proposal, Ownership Domains [2], and its implementation in a concrete language, AliasJava [5]. Unlike some of the other approaches which are only paper designs, AliasJava has had a publicly available open-source compiler for a few years [4]. In addition, it has been evaluated on actual object-oriented programs (see case studies in [5] and [1]). The Universes system is another ownership-based system we are aware of that has been applied in a case study involving a non-trivial system [12].

2 AliasJava

AliasJava [5] is a concrete implementation of the Ownership Domains system proposed by Aldrich et al [2]. AliasJava extends Java to express how data is confined within, passed among, or shared between components and objects in a software system: developers can express controlled aliasing through ownership domains and the lack of aliasing through uniqueness using annotations on reference types.

Ownership Domains. AliasJava controls aliasing relationships in object-oriented programs by dividing objects into conceptual groups called ownership domains and allowing architects to specify high-level policies that govern references between ownership domains. AliasJava supports abstract reasoning about data sharing by assigning each object in the system to a single ownership domain. There is a top-level ownership domain denoted by the keyword `shared`. In addition, each object can declare one or more domains to hold its internal objects, thus supporting hierarchical specifications. AliasJava has a keyword `domain` to define ownership domains. Figure 2 uses a `Sequence` abstract data type to illustrate the ownership model used in AliasJava. The `Sequence` object is part of a top-level `owner` ownership domain. Within a `Sequence` object, the `iters` ownership domain is used to hold iterator objects that clients use to traverse the sequence, and the `owned` ownership domain is used to hold the `Cons` cells in the linked list that is used to represent the sequence. The first few lines of code within the `Sequence` class (Figure 3) declare the `owned` and `iters` domains, link specifications between these two domains (the `link` construct) and a reference to the head of the list. By default, each object has a private domain called `owned`, so the corresponding domain declaration can be omitted.

Domain Permissions. Objects within a single ownership domain can refer to one another, but references can only cross domain boundaries if the programmer specifies an architectural link between the two domains when they are created. Each object can declare

```
class JavaClass {
    private List signers;

    public List getSigners() {
        return this.signers;
    }
}
// (Malicious) clients can mutate signers field!
class MaliciousClient extends ... {
    public void addTrojanHorse(JavaClass c)
    {
        List signers = c.getSigners();
        signers.add( this );
    }
}
```

Figure 1: In an early version of the JDK, the `Class.getSigners` method returned the internal list of signers rather than a copy, allowing untrusted clients to pose as trusted code by modifying the `signers` object through its alias.

a policy describing the permitted aliasing among objects in its internal domains, and between its internal domains and external domains. AliasJava supports two kinds of policy specifications:

- A link from one domain to another, denoted with a dashed arrow in the diagram, allows objects in the first domain to access objects in the second domain;
- A domain can be declared public. Permission to access an object automatically implies permission to access its public domains.

For example, in Figure 2 the `Sequence` object declares a link from its `iters` domain to its `owned` domain, allowing the iterators to refer to objects in the linked list. The `iters` domain is public, allowing clients to access the iterators, but the `owned` domain is private so clients must access the elements of `Sequence` through its iterator interface rather than traversing the linked list directly. In addition to the explicit policy specifications mentioned above, AliasJava includes the following implicit policy specifications:

1. An object has permission to access other objects in the same domain;
2. An object has permission to access objects in the domains that it declares.

The first rule allows the different `Cons` cells in the linked list to access each other, while the second rule allows the sequence to access its iterators and linked list. Any reference not explicitly permitted by one of these rules is prohibited, according to the principle of

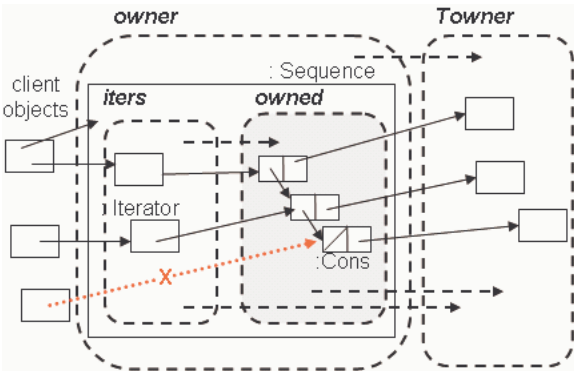

Figure 2: A conceptual view of the ownership and aliasing model in AliasJava. The rounded, dashed rectangles represent ownership domains, with a gray fill for private domains, and no fill for public domains. Solid rectangles represent objects. The top-level `shared` domain contains the highest-level objects in the program. Each object may define one or more domains that in turn contain other objects. Dashed arrows represent link permissions between domains.

least privilege. It is crucial that there is no transitive access rule: for example, even though clients can refer to iterators and iterators can refer to the linked list, clients cannot access the linked list directly because the sequence has not given them permission to access the `owned` domain. Thus, the policy specifications allow developers to specify that some objects are an internal part of an abstract data type's representation, and the compiler enforces the policy, ensuring that this representation is not exposed.

Domain Parameters. Figure 3 shows how the `Sequence` Java code can be annotated with aliasing information to model the constraints expressed in Figure 2. AliasJava uses the Java 1.5 type parameters syntax to define domain parameters (e.g., `class Sequence`$<$ *Towner* $>$) as well as binding actuals to formals (e.g., `Sequence`$<$ `owned` $>$ `seq = new Sequence<owned>();`). The `head` field is of type `owned Cons<Towner>`, denoting a `Cons` linked list cell that resides in the `owned` domain and holds an object that resides in the `Towner` domain. The `add` member function constructs a new `Cons` cell for the object passed as argument and adds it to the head of the list. Skipping ahead to the definition of the `Cons` cell class, we see that it is also parameterized by the domain parameter `Towner`. The class contains a field `obj` holding an element in the list, along with a `next` field referring to the next `Cons` cell (or null, if this is the end of the list). The `next` field has type `owner Cons<Towner>`, indicating that the next cell in the list has the same owning domain as the current cell (i.e., all the cells are part of the `Sequence`'s `owned` domain).

Domain Hierarchy. The set of named ownership domains each object declares are nested within the domain that owns the object, so ownership defines a forest of trees where each parent owns its children and the roots of the tree are unique. Unique objects may be assigned to an ownership domain, attaching one ownership tree as a subtree of another. All connected components must be part of an ownership domain declared by the component

```
class Sequence<Towner> assumes owner -> Towner {
  domain owned;
  public domain iters;
  link owned -> Towner;
  link iters -> Towner, iters -> owned;
  owned Cons<Towner> head;

  void add(Towner Object o) { head = new Cons<Towner>(o,head); }
  iters Iterator<Towner> getIter() {
    return new SequenceIterator<Towner, owned>(head); }
}

class Cons<Towner> assumes owner -> Towner {
  Cons(Towner Object obj, owner Cons<Towner> next)
    { this.obj=obj; this.next=next; }
  Towner Object obj;
  owner Cons<Towner> next;
}

interface Iterator<Towner> {
  Towner Object next();
  boolean hasNext();
}

class SequenceIterator<Towner, list> implements Iterator<Towner>
          assumes list -> Towner {
  private list Cons<Towner> current;
  public SequenceIterator(list Cons<Towner> head) {current = head; }
  public boolean hasNext() { return current != null; }
  public Towner Object next() {
    Towner Object obj = current.obj;  current = current.next; return obj; }
}

public class SequenceClient {
  domain state;
  final state Sequence<state> seq = new Sequence<state>();
  void doSomething(state Object o) { System.out.println("Iterated on " + o);  }
  public void run() {
    state Object obj = new Integer(5);
    seq.add(obj);
    seq.add(new Integer(7));

    seq.iters Iterator<state> i = this.seq.getIter();
    while (i.hasNext()) {
      state Object cur = i.next();
      doSomething(cur);
    }
  }
}
```


Figure 3: Sequence example in AliasJava. Adapted from the AliasJava distribution [4].

```
class JavaClass {
     private owned List signers;

     private owned List getSigners() {
        return this.signers; }

     public void foo() {
        lent List x = this.getSigners();
        // do stuff using x
    }
}
```

Figure 4: AliasJava re-implementation of the JavaClass: the list is declared `owned`; the type checker now requires `getSigners()` to be marked private, since a public method may not have owned in its signature. Clients can only call the public method `foo()`.

making the connection.

Unique Data. While ownership is useful for representing persistent aliasing relationships, it cannot capture the common scenario of an object that is passed between objects without creating persistent aliases. Objects to which there is only one reference (including newly-created objects) are annotated `unique` in AliasJava. Unique objects can be passed from one ownership domain to another, as long as the reference to the object in the old ownership domain is destroyed when the new reference is created.

Lent Data. We also allow one ownership domain to temporarily lend an object to another ownership domain, with the constraint that the second ownership domain will not create any persistent references to the object. For example, we annotate a method parameter as `lent` to indicate that it is a temporary alias. A `unique` object can be passed to a method as a `lent` argument even without destroying the original unique reference. The method can pass on the object as a `lent` argument to other methods, but cannot return it or store it in a field. Using `lent`, we can also 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. Thus, the `lent` annotation preserves all of the reasoning about the `unique` object, but adds a large amount of practical expressiveness.

Shared Data. Objects marked with the `shared` annotation may be aliased globally. Unfortunately, little reasoning can be done about `shared` references, except that they may not alias non-`shared` references. However, `shared` references are essential for interoperating with existing runtime libraries, legacy code, and static fields, all of which may refer to aliases that are not confined to the scope of any object instance.

AliasJava Example. Figure 4 shows one possible fix for the aliasing bug identified earlier using the `owned` annotation. Figure 5 shows another possible fix for the aliasing bug identified earlier, using domain parameters.

```
class JavaClass<data> {
    private owned List signers;

    public data List getSigners() {
        data List copy = new List();

        for(int i = 0; I < this.signers.size();i++)
           copy.add(this.signers.get(i));
        return copy;
    }
}
```

Figure 5: AliasJava re-implementation of the JavaClass, returning a copy of the list of signers in the domain identified by the domain parameter `data`. Alternatively, the copy could have been returned in the global `shared` domain to avoid the need for a parameter.

3 JavaD: AliasJava with annotations

JavaD re-implements the AliasJava language and analysis as annotations using the annotation facility in Java 1.5 [17]. In particular, Java programs with JavaD annotations are legal Java 1.5 programs unlike AliasJava programs which are no longer legal Java programs.

3.1 Motivation

AliasJava is currently implemented using a modified version of the Barat infrastructure [6][1]. JavaD re-implements the AliasJava language and analysis using the Eclipse Java Development Tooling (JDT) [20] infrastructure and the Crystal data flow analysis framework [3].

Since AliasJava is a non-backwards compatible extension to the Java programming language, AliasJava programs have only basic tool support available to them. Re-implementing the language and the analysis as annotations improves the adoptability of the ownership domains technique by mainstream Java developers as follows:

- **Improved tool support**: all the capabilities of the Eclipse integrated development environment become available to AliasJava programs, from advanced debugging capabilities to refactoring to syntax highlighting;
- **Ease of extensibility**: using annotations would make it easier to extend the AliasJava language in a non-breaking way. Some of the candidate annotations include ones for external uniqueness [10] and read-only references [15] among others;

[1] Although open-sourced, the Barat infrastructure [6] is not maintained at the same level as the Eclipse open source project which is much larger. In particular, the Barat infrastructure still does not support Java 1.5.

- **Support for incrementality**: using annotations gives the ability to *incrementally* and *partially* specify annotations. This is necessary for dealing with large code bases and would enable us in turn to conduct case studies to evaluate ownership domains on large real-world object-oriented programs.

Since the main purpose of this project is to address the adoptability of AliasJava, usability is a primary consideration. Although annotations may be more verbose than an elegantly designed language, we tried to make JavaD annotations as usable as possible, using the following strategies:

- **Only warnings**: the analysis generates only warnings and no errors about inconsistent annotations;

- **Reasonable defaults**: we supply reasonable defaults to reduce the annotation burden. We reuse the same defaults as AliasJava;

- **Consistency:** only using annotation imposes several restrictions on a language designer. For instance, Java 1.5 allows annotations only on declarations. In particular, there may be some cases where an extra temporary variable may need to be declared, just for the purpose of adding annotations to it. This will be discussed further for new expressions, cast expressions (both implicit and explicit) and arguments for method/constructor invocation.

- **Non-executable annotations:** finally, we use entirely non-executable annotations, i.e., the behavior of the code is not affected by annotations. Our approach involves purely a static analysis and does not interfere with the running of the program: in particular, unlike AliasJava, where there may be some runtime exceptions[2]related to bad casts, programs annotated with JavaD behave in exactly the same way as they did before, since the annotations have no effect whatsoever at runtime. As a result, however, the system is unsound at casts.

3.2 Annotation Design

We have defined the following Java 1.5 annotations. For maximum flexibility, all annotation values are strings. Annotations that are plural take values that are arrays of strings[3]. We describe the annotations and illustrate each one with code snippets from the Sequence example (the full example is shown in Figure 6).

@Domain: Specify the actual annotation, the actual parameters and the actual array parameters on a variable, field, method return type, method parameter.

[2]AliasJava has the option of generating `.java` files from annotated `.archj` files; in that case, the generated Java files contain additional runtime checks inserted by the code generator.

[3]Java 1.5 allows the following syntactic sugar: single-element array-valued single-member annotation can be written without the curly braces { ... }, e.g., `@Domains("iters")` or `@Domains({"iters"})`. Otherwise, the array syntax requires curly braces, e.g., `@Domains({"iters", "owned"})`.

- **Format**: $parameter < parameter, \ldots > [parameter, \ldots]$ where
  - $parameter$ can be any alias annotation or refer to a public domain of an object, e.g., $seq.iters$;
  - $< parameter, \ldots >$ (angle brackets) optional annotation for the ordered list of actual domain parameters;
  - $[parameter, \ldots]$ (square brackets) optional annotation for the ordered list of actual array parameters, by order of array dimension.
- **Applies to**: parameter, field, local variable, method, constructor[4]
- **Examples**:

```
@Domain("unique<Towner, owned>")
SequenceIterator sequenceIterator = new SequenceIterator(head);
...

public static void main(@Domain("lent[shared]")String args[]) {
...
}
```

@Domains: declare domains on a type (class or interface).

- **Format**: *name*
- **Applies to**: type (class or interface)
- **Examples**:

```
@Domains({"iters"})
class Sequence
```

@DomainParams: declare domain parameters on a type (class or interface)

- **Format**: *name*
- **Applies to**: type (class or interface)
- **Examples**:

```
@DomainParams({"Towner"})
class Sequence
```

@DomainInherits: pass parameters to superclass or implemented interfaces

- **Format**: $typename < parameter, \ldots >$
- **Applies to**: type (class or interface)
- **Examples**:

```
@DomainParams({"Towner", "list"})
@DomainInherits({"Iterator <Towner>"})
class SequenceIterator implements Iterator {
  ...
}

@DomainParams({"Towner"}) {
interface Iterator {
   ...
}
```

@DomainReceiver: declare annotation on the receiver.

- **Format**: *name*
- **Applies to**: constructor or method
- **Examples**:

3.3 Examples

In this section, we illustrate how to annotate an abstract data type with an interator, an interesting benchmark for ownership systems. Figure 3 shows the original AliasJava program. Figure 6 shows snippets from the equivalent JavaD program.

4 Tool Design and Implementation

The tool design and implementation has been heavily inspired from the original AliasJava compiler [4]. The analysis consists of:

- **Annotation management:** associate with each AST node an annotation value;
- **First-Pass (visitor-based) Analysis:** retrieve the explicit annotations from the AST nodes (for types, variables, and methods) and propagate them to expressions;
- **Second-Pass (visitor-based) Analysis:** check the annotations on each expressions using the AliasJava rules.

We discuss each one in turn.

[4]We use the $@Target$ feature, e.g., $@Target(ElementType.PARAMETER, ...)$ to specify where a specific annotation is allowed (in this case, on constructor or method parameters).

```
@Domains({"iters"}) @DomainParams({"Mowner", "Towner"})
class Sequence {
  @Domain("owned<owned,Towner>") Cons head;
  void add(@Domain("Towner")Object o) {
      head = new Cons(o,head); }
  @Domain("iters<Towner>") Iterator getIter() {
      @Domain("unique<owned, Towner, owned>")
      SequenceIterator sequenceIterator = new SequenceIterator(head);
      return sequenceIterator;
  }
}
@DomainParams({"Mowner", "Towner"}) class Cons {
  Cons(@Domain("Towner")Object obj,@Domain("Mowner <Mowner, Towner>")Cons next) {
     this.obj=obj; this.next=next; }
  @Domain("Towner") Object obj;
  @Domain("Mowner<Mowner, Towner>") Cons  next;
}
@DomainParams({"Towner"}) interface Iterator {
    @Domain("Towner") Object next();
    boolean hasNext();
}
@DomainParams({"Mowner", "Towner", "list"}) @DomainInherits({"Iterator<Towner>"})
class SequenceIterator implements Iterator {
  @Domain("unique")SequenceIterator(@Domain("list<Mowner, Towner>") Cons head) {
      current = head; }
  @Domain("list <Mowner, Towner>")
  private Cons current;
  public boolean hasNext() { return current != null; }
  public @Domain("Towner") Object next() {
      @Domain("Towner")Object obj = current.obj;
      current = current.next;
      return obj;
}
@Domains({"state"})
 public class SequenceClient {
    final @Domain("state<owned, state>") Sequence seq = new Sequence();
    void doSomething(@Domain("state")Object o) { ... }
    public void run() {
        @Domain("state")Object obj = new Integer(5);
        seq.add(obj);
        @Domain("unique")Integer int7 = new Integer(7);
        seq.add(int7);
        @Domain("seq.iters<state>") Iterator i = this.seq.getIter();
        while (i.hasNext()) {
            @Domain("state") Object cur = i.next();
            doSomething(cur);
        }
    }
}
```

Figure 6: Sequence example in with JavaD annotations (with explicit “owner”).

4.1 Annotation Information

For each AST node, the tool maintains the following information:

- **Annotation:** represents the ownership domain (e.g., `owned` or `seq.iters`) or the alias annotation on a reference type (e.g., `lent`);
- **Parameters:** represent the formal or actual domain parameters on a reference type;
- **ArrayParameters:** represent the annotations on each array dimension for array types;
- Mapping from formals to actuals.

4.2 First-Pass Analysis

The first pass is visitor-based[5]analysis to perform the following:

Identify Problematic Patterns. During this pass, we identify problematic code patterns that will need to be replaced with equivalent constructs, namely by declaring a local variable and adding the appropriate annotations to it[6].

Read Annotations from AST. The Java 1.5 annotations that are added to a program become part of the AST. An AST visitor locates these nodes in the AST and parses their contents[7]. In addition, the visitor infers default annotations for some program constants that cannot be annotated, e.g., it infers `unique` on NullLiteral AST nodes and StringLiteral AST nodes. The annotations that are read are stored in a hash table mapping each AST node to an annotation structure. This mapping is used by the second pass analysis to check the correctness of the annotations.

Propagate Local Annotations. The AST visitor also propagates annotations to all the expression nodes in the AST by translating formals to actuals. This visitor visits AST nodes corresponding to:

- ArrayAccess: handle array access expressions
- ArrayCreation: mark array creation expressions as `unique`
- ArrayInitializer: mark array initializer expressions as `unique`
- Assignment

[5]This visitor must be a post-order visitor in order to correctly check expressions such as `Iterator i = this.seq.getIter()`. Post-ordering will ensure that the correct annotation for the FieldAccess (`this.seq`) is generated before that of the MethodInvocation (`getIter()`).

[6]Using the Eclipse built-in refactoring ("Extract Local Variable"), this operation can be performed with very little effort.

[7]We used JavaCC [14] to generate a small parser for the annotation string when it can get complex (e.g., as in the `@Domain` case). In most other cases, we used simple string manipulation in Java.

- CastExpression: check for unsupported constructs
- ClassInstanceCreation: check for unsupported constructs
- FieldAccess
- ConditionalExpression: propagate annotations for conditional expressions, i.e., `expression ? thenExpression : elseExpression;`
- InfixExpression: mark string concatenations (using infix + operator) as `unique`
- MethodDeclaration: retrieve annotation from node
- MethodInvocation: translate formals to actuals, and store for expression
- NullLiteral: mark the null constant as `unique`
- QualifiedName: propagate annotation, translate formals to actuals, and store for expression
- ReturnStatement: check for unsupported constructs
- SingleVariableDeclaration: retrieve annotation from node
- StringLiteral: mark string constants as `unique`
- ThisExpression
- TypeDeclaration: retrieve annotation from node
- VariableDeclarationFragment: retrieve annotation from node

4.3 Second-Pass Analysis

Check Rules. The second pass also consists of an AST Visitor to check the AliasJava rules. This visitor visits the following nodes and checks the corresponding rules:

- TypeDeclaration: inheritance rules
- FieldDeclaration: declaration rules
- SingleVariableDeclaration: declaration rules
- VariableDeclarationFragment: declaration rules
- MethodDeclaration: check method rules
- Assignment: check assignment, initializers

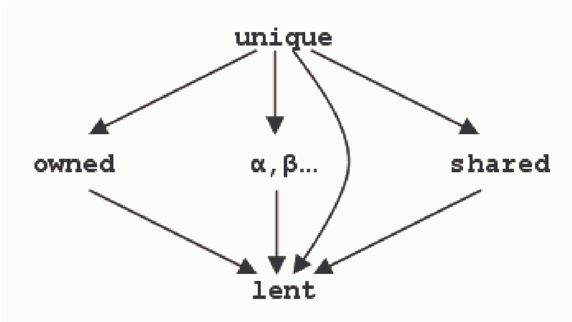

Figure 7: Arrow means data can flow between variables with two annotations.

- ClassInstanceCreation: constructor rules
- MethodInvocation: method call rules
- ReturnStatement: check assignment rules (of return value)
- FieldAccess: assignment

Value flow analysis Checking the assignment rule requires a value flow analysis, based on the following rules in AliasJava:

- A variable with any type annotation can be assigned a unique value;
- lent variables can be assigned a value with any type annotation;
- Values with type annotations owned and shared as well as declared domains are kept separate from each other.

The value flow analysis cannot be implemented as a regular dataflow analysis because it does not correspond to a lattice (See Figure 7). We reused the Live Variables Analysis (LVA) from the Crystal Data Flow analysis framework. LVA is invoked intra-procedurally at each method boundary using a separate visitor that looks for a variable with a specific binding information.

5 Evaluation

We tested and evaluated the JavaD tool on the following examples.

5.1 AliasJava Examples

We tested the JavaD tool by taking some of the examples provided with the ArchJava distribution [4] and converting them from AliasJava syntax to Java syntax with JavaD annotations.

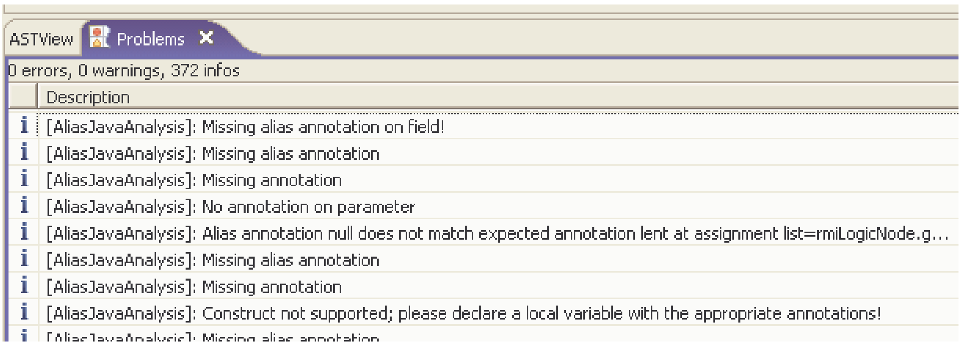

Figure 8: The Eclipse Problems window showing the JavaD warnings on the raw Courses example.

5.2 Courses Example

We also annotated a small Java program that corresponds to a three-tier course registration system. The tool identified several instances of unsupported coding patterns discussed later, for which we used the Eclipse refactoring ("Extract Local Variable") to replace them with supported coding patterns. We also used the Eclipse Find/Replace feature to replace all instances of `String` with `@Domain("shared")String`. In addition, we replaced all uses of `java.util.ArrayList` with the annotated `Sequence` Abstract Data Type, and used the Iterator construct to iterate over the elements of the collection.

During the annotation process, we frequently invoked the JavaD analysis. We initially started out with several hundreds of warnings (See Figure 8), but that number was brought down very quickly. The tool was helpful in pinpointing incorrect annotations. The most common annotation warnings were missing alias annotations, not passing the appropriate domain parameters and forgetting to mark variables (used in annotations) as final. In the final program, there are some remaining expected[8]warnings regarding missing annotations on return types of various library functions on `java.lang.String`.

Annotating the courses example took a couple of hours, not including several interruptions to fix various problems in the implementation of the tool.

6 Limitations and Future Work

In this section, we discuss some of the current limitations of this work and our plans for future enhancements.

[8]Currently, the tool does not support annotating library code. Eliminating these warnings requires adding annotations to various methods on the `java.lang.String` class defined in the Java Standard Library.

6.1 Missing Features

Due to time constraints, the following AliasJava language features were not implemented:

- Support for the `owner` annotation (the current support in the AliasJava compiler is broken). The original Sequence example shown earlier did make use of it (See Figure 3). However, we rewrote it by passing the `Mowner` as an explicit domain parameter (by convention, we use the first one), as in Figure 6);
- Different checks for public and private domains;
- Support for method domain parameters (already supported in AliasJava); such methods are typically useful in library code.
- Sanity checks for the annotations: e.g., check that the `@DomainInherits` does not reference a nonexistent superclass or an interface that is not implemented; disallow the standard alias types (`owned`, `lent` or `unique`) as private domains in the `@Domains` annotation;
- Domain link specifications (`@DomainLinks` and `@DomainAssumes`) (already supported in AliasJava) and the associated checks (currently, partially implemented).

Some tool-specific future work might include:

- Creating an Eclipse builder so the analysis is invoked automatically as the program is modified;
- Adding the ability to suppress certain warnings; and in turn, add the ability to re-run the analysis without any suppressions to make sure that developers are not simply suppressing warnings instead of fixing legitimate errors;
- Improving cryptic error messages: e.g., a message such as "Alias parameters [] of new Course(objCourseFile.readLine()) instance don't match [objectsDom] alias parameters of receiver at call ..." (for the "Before" construct in Figure 9) is a roundabout way of telling the developer that she must simply declare two local variables and add annotations to them;
- Adding the ability to turn off defaults: for program comprehension, we think that having all annotations spelled out is preferable since a developer does not have to understand both the significance of the presence of an annotation, as well as the absence thereof[9].

[9] There was a time when, in some unsafe languages, it was possible to have variables without type annotations, and the type was assumed to be some default; thankfully, this is now a passing trend.

6.2 Limitations of Java 1.5 annotations

We ran into several of the limitations of Java 1.5 annotations. Some of the limitations include:

1. A declaration cannot have multiple annotations of the same annotation type;
2. Annotation types cannot have members of the their own type;
3. It is only legal to use single-member annotations for annotation types with multiple members, as long as one member is named `value`, and all other members have default values. Otherwise, the more verbose syntax is required, e.g., `@Name(first = "Joe", last = "Hacker")`;
4. Annotation types cannot extend any entity (class, interface or annotation).

The fist restriction prevented us from using the `@Domain` annotation to specify both the annotation on the receiver and on the return type of a method.

The second restriction prevented us from having shorthand constant annotations for some of the frequently used ones, e.g., `@owned` instead of `@Domain("owned")`. However, such constants cannot be used in annotations such as `@Domain(annotation = @owned, parameters = {@owned})`. To avoid having multiple ways of performing the same thing, in the end, we resorted to using strings and doing our own parsing, as in annotations of the form `@Domain("owned <owned>")`. This means that developers may be more likely to introduce minor mistakes by misspelling standard annotations since the auto-complete feature is not available for free-form strings. However, the analysis will catch such problems early enough.

The third restriction, i.e., the lack of positional arguments, prevented us from expressing constructs of the form `@Domains({"public1", "public2"}, {"private1", "private2"})`. Instead, we have to use the more verbose syntax of `@Domains(publicDomains = {"public1", "public2"}, privateDomains = {"private1", "private2"})` or `@Domains({"public1", "public2"}, privateDomains = {"private1", "private2"})`.

Finally, Java 1.5 annotations can only be added to declaration points. In particular, we found two kinds of expressions to be problematic, namely new expressions (See Figure 9) and cast expressions (See Figure 10). As a result, some legal Java constructs will need to be replaced with equivalent but more verbose constructs that involve declaring local variables and adding the appropriate annotations to them.

As an an alternative, we could have used stylized comments, but we think that comments would have been even less structured and easier to overlook by developers than annotations. Most code editors, including the Eclipse Java editor, highlight annotations differently than comments[10]. Furthermore, the stylized comments are bound to look like commented out code which goes against many coding guidelines.

[10] We are not aware of empirical evaluations that evaluated whether comments or annotations help or hinder program comprehension.

```
Before:
while (objCourseFile.ready()) {
    this.vCourse.add(new Course(objCourseFile.readLine()));
}
After:
while (objCourseFile.ready()) {
    @Domain("shared")String line = objCourseFile.readLine();
    @Domain("objectsDom <objectsDom>")Course course = new Course(line);
    this.vCourse.add(course);
}
```

Figure 9: Re-writing a new expression by declaring a few local variables with the appropriate annotations.

```
Before:
ArrayList vCourse = objStudent.getRegisteredCourses();
for (int i=0; i<vCourse.size(); i++) {
    if (((Course) vCourse.get(i)).conflicts(objCourse)) {
        lock.releaseLock();
        return "Registration conflicts";
    }
}
After:
@Domain("lent")ArrayList vCourse = objStudent.getRegisteredCourses();
for (int i=0; i<vCourse.size(); i++) {
    @Domain("lent<objectsDom>")Course course = (Course) vCourse.get(i);
    if (course.conflicts(objCourse)) {
        lock.releaseLock();
        return "Registration conflicts";
    }
}
```

Figure 10: Re-writing a cast expression by declaring a local variable.

6.3 Future Work

There are several directions that we can pursue next.

6.3.1 Standard and Third-Party Libraries

We would like to support adding annotations to the standard JDK libraries and other third-party libraries. There are two approaches: one that involves annotating the library and one that involves placing annotations in separate files. The latter is the preferred approach since it does not require making changes to library or third-party code which may not be available, and when it is, tends to evolve separately.

Even when storing annotations in separate files, it may be necessary to reduce the annotation burden, e.g., by using pattern matching to annotate all parameters or return types of type `java.lang.String` with `@Domain("shared")`.

6.3.2 Additional Alias Types

As discussed earlier, an important goal of this project is to support extensibility. In particular, we hope to develop new kinds of annotations to support "external uniqueness" [10], "readonly" references [15] among many possible ones. In particular, an "immutable" annotation would be useful for annotating objects of type `String`.

6.3.3 Interactive Annotation Inference

Annotating existing code is difficult and time-consuming, since one has to first determine the appropriate ownership parameters, and annotate almost every line of code with a reference type (assuming the default is not suitable). We hope this tool will serve as a good starting point to infer annotations interactively. In particular, Eclipse provides the functionality to graphically preview refactorings (in this case, the addition of annotations) before they are actually applied to the program.

One interactive annotation inference tool based purely on a static analysis that we are aware of [11] makes use of additional annotations to guide the inference algorithm: for instance, the user can supply an `@Complete` annotation as a hint to the inference algorithm to indicate that the list of ownership parameters may not be extended. We are also thinking about using `@Suggest(...)` annotations to let the user provide various hints to the inference algorithm or to have the algorithm generate its suggestions as `@Suggested(...)` annotations. One of the benefits of using annotations over language extensions is that an inference algorithm can in no way break an existing program by inserting incorrect annotations.

6.3.4 Warning Suppression

Cooper mentions how "certain references are not used in ways that could cause aliasing bugs and could therefore be considered harmless aliases. It may be possible to extend AliasJava to incorporate an annotation for such references" [11]. Based on our own experience [1], we agree that it might be useful to have escape mechanisms to tolerate small local inconsistencies and to suppress warnings on innocuous code.

6.3.5 Warning Suppression

AliasJava does not allow implementations of interface methods or overrides of abstract methods to change the ownership annotation. Allowing subtypes to be annotated in ways that are incompatible with their supertypes breaks substitutability and is unsound. However, two applications instantiating a framework, library or third-party component, may need to annotate the same methods in incompatible ways depending on usage. For instance, AliasJava annotates the `this` receiver by default with `lent`. However, application code implementing abstract methods or interface methods (e.g., from a user interface framework) can do arbitrary things with the `this` pointer such as storing it or passing it to other components. So when annotating legacy GUI code, the `lent` default may not be appropriate. We think this problem can be solved by adding some kind of parameter to the superclass, e.g., by making each method parametric in the ownership type of its receiver.

6.3.6 Generic Ownership

We have ported the AliasJava language and analysis to the Eclipse infrastructure and made use of a Java 1.5 language feature, which makes it possible, at least in principle, to also handle generics. However, due to time constraints, we did not study the implications of using Java 1.5 generics with JavaD annotations, although the *Sequence* example would be a prime candidate.

It may very well be the case that adding ownership annotations to Java 1.5 with generics will be too verbose. A promising alternative approach is to combine ownership and generic types, as is being proposed by Potanin et al [16]. This is an area that we will be watching closely.

7 Conclusion

We presented a re-implementation of the AliasJava language and analysis as a set of Java 1.5 annotations, using the Eclipse Java Development Tooling (JDT) infrastructure and the Crystal Data Flow Analysis framework.

We think this tool can encourage additional case studies to first annotate and then maintain real object-oriented implementations to evaluate the true benefits of using ownership domains for program understanding and evolution. This tool could also spur future activity by making it easier to extend the language with richer annotations and build an interactive annotation inference tools that do not break existing programs.

References

[1] Marwan Abi-Antoun, Jonathan Aldrich and Wesley Coelho. A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing. In Journal of Systems and Software *(To Appear)*. 2006.

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

[3] Jonathan Aldrich and David Dickey. The Crystal Data Flow Analysis Framework. `http://www.cs.cmu.edu/~aldrich/courses/654/`

[4] Jonathan Aldrich. ArchJava Downloads. `http://www.archjava.org/`

[5] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. Proc. Object Oriented Programming Systems, Languages and Applications, Seattle, Washington, November 2002.

[6] Boris Bokowski and Andr Spiegel. BaratA Front-End for Java. Freie Universitat Berlin Technical Report B-98-09, December 1998.

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

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

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

[10] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. Proc. European Conference on Object-Oriented Programming, Darmstadt, Germany, July 2003.

[11] Will Cooper. Interactive Ownership Type Inference. School of Computer Science Senior Thesis, Carnegie Mellon University. 2005.

[12] Thomas Hächler. Applying the Universe type system to an industrial application: case study. Master Project Report, Departement of Computer Science, Swiss Federal Institute of Technology, 2005.

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

[14] JavaCC. Available at `https://javacc.dev.java.net/`

[15] Peter Müller 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.

[16] Alex Potanin, James Noble, David Clarke and Robert Biddle. Featherweight Generic Ownership. In 7th Workshop on Formal Techniques for Java-like Programs - FTfJP'2005.

[17] Sun Microsystems, Inc. Java Specification Requests JSR 175: A Metadata Facility for the JavaTM Programming Language.

[18] Eclipse Java Development Tooling (JDT) core. `http://dev.eclipse.org/viewcvs/index.cgi/jdt-core-home/main.html?rev=1.97`

[19] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`

[20] Eclipse Java Development Tooling (JDT) core. `http://dev.eclipse.org/viewcvs/index.cgi/jdt-core-home/main.html?rev=1.97`

[21] Werner Dietl and Peter Muller. Exceptions in Ownership Type Systems. In E. Poll, editor, Formal Techniques for Java-like Programs, pp. 4954, 2004.