JavaD: Bringing Ownership Domains to Mainstream Java

Marwan Abi-Antoun [a]

[a] *Institute for Software Research Intl (ISRI)*
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213

Abstract

Ownership types have been receiving much attention from the research community. However, few of the proposed designs have been implemented or evaluated on real object-oriented implementations. AliasJava has been available for a few years and has been applied on several case studies.

Currently, AliasJava is implemented as a non-backwards compatible extension of the Java programming language. As a result, none of the tool support for Java (from debugging and refactoring to syntax highlighting) 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, this makes it harder specify the ownership and aliasing annotations for a large legacy system, since the program cannot be annotated partially and incrementally with AliasJava.

We present 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 some lessons learned and future plans.

1 Introduction

"The big lie of object-oriented programming is that objects provide encapsulation" [12]. 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

```
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 );
    }
}
```

Fig. 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.

support (see [7] for a good survey), and proposed several solutions, e.g., Islands [12] and Universes [14] to name only a few.

We look in particular at one proposal, Ownership Domains [1], and its implementation in a concrete language, AliasJava [4]. Unlike some of the other approaches, which are only paper designs, AliasJava has had a publicly available open-source compiler for a few years [3]. In addition, it has been evaluated on actual object-oriented programs (see case study in [4]). The Universes system is the only other ownership-based system we are aware of that has been applied in a case study involving a non-trivial system [11].

2 AliasJava

AliasJava [4] is a concrete implementation of the Ownership Domains system proposed by Aldrich et al. AliasJava has a keyword `domain` to define ownership domains. By default, each object has a private domain called `owned`. Figure 2 shows one possible fix for the aliasing bug identified earlier using the `owned` annotation. AliasJava also 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`). Figure 3 shows another possible fix for the aliasing bug identified earlier, using domain parameters. AliasJava also provides a small number of standard aliasing annotations to describe data that is:

```
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
    }
}
```

Fig. 2. 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()`.

```
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;
    }
}
```

Fig. 3. 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.

- Confined with an object (`owned`) (default domain)
- Passed linearly from one object to another (`unique`)
- Shared temporarily (`lent`) within a method invocation
- Shared persistently (`shared`) globally

3 JavaD: AliasJava with annotations

JavaD re-implements the AliasJava language and analysis as annotations using the annotation facility in Java 1.5 [16]. 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 [5] [1] . More specifically, we wanted to re-implement the AliasJava language and analysis using the Eclipse Java Development Tooling (JDT) [19] infrastructure, and the Crystal data flow analysis framework [2].

Since it is a non-backwards compatible extension to the Java programming language, AliasJava programs have only basic tool support available to them. We think 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 add extensions to the AliasJava language in a non-breaking way. Some of the candidates include external uniqueness [9] and read-only references;
- **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:

- **Generate only warnings**: the analysis only generates warnings and does not generate errors about inconsistent annotations;
- **Use reasonable defaults**: we supply reasonable defaults to reduce the annotation burden. We reuse the same defaults as AliasJava;
- **Be consistent:** when using only annotations, there are several restrictions that a language designer has to deal with, such as only being able to add annotations to variable declarations. In particular, there may be some cases where an extra temporary variable may need to be declared, just for the sake of adding annotations to it. This point will be revisited when we talk about new expressions, cast expressions (both implicit and explicit), and method/constructor invocation.
- **No runtime checks**: finally, the approach involves purely a analysis static

[1] Although open-sourced, the Barat infrastructure [5] is not maintained at the same level as the Eclipse much larger open source project. In particular, the Barat infrastructure does not support Java 1.5.

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. The annotations have no effect whatsoever at runtime.

3.2 Annotation Design

We have defined the following Java 1.5 annotations. For maximum flexibility, all annotation values are strings or arrays of strings. All the annotations that are plural take an array of strings [3]. The annotations are illustrated by examples from the fully annotated Sequence client example.

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

- **Format**: *parameter* < *parameter*, . . . > [*arrayParameter*, . . .] where
  - *parameter* can be any alias annotation, or refer to the public domain of an object, e.g., *seq.iters*;
  - < *parameter*, . . . > (angle brackets) optional annotation for the ordered list of actual domain parameters;
  - [*parameter*, . . .] (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"})
```

[2] AliasJava has the option of generating .java files from annotated .archj files; in that case, the .java files will have the additional runtime checks.

[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 `@Domain({"iters"})`. The array syntax requires curly braces, e.g., `@Domain({"iters", "owned"})`

```
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 super class and implemented interfaces

- **Format**: *typename* < *parameter*, ... >
- **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, which is an important benchmark for flexible ownership systems. Figure 5 shows the original AliasJava program. Figure 6 shows snippets from the equivalent JavaD program.

[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).

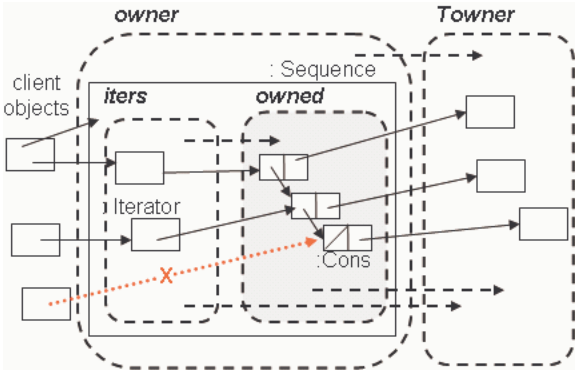

Fig. 4. The Sequence abstract data type uses a linked list for its internal representation. A public `iters` domain holds its iterators. A private `owned` domain holds the linked list. Link declarations specify that iterator objects in the `iters` domain have permission to access objects in the `owned` domain; outside objects cannot directly access `Cons` cells. Both domains can access the domain parameter `Towner`.

4 Tool Design and Implementation

The tool design and implementation has been heavily inspired from the original AliasJava compiler [3]. 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.

4.1 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.

```
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);
    }
  }
}
```


Fig. 5. Sequence example in AliasJava. Adapted from the AliasJava distribution [3].

```
@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);
        }
    }
}
```


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

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. The visitor locates these nodes in the AST and parses their contents [7]. n 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 vists AST nodes corresponding to:

- ArrayAccess: handle array access expressions
- ArrayCreation: mark array creation expressions as `unique`
- ArrayInitializer: mark array initialize expressions as `unique`
- Assignment
- CastExpression: check for unsupported constructs
- ClassInstanceCreation: check for unsupported constructs
- FieldAccess
- ConditionalExpression: propagate annotations for condition 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 array initialize expressions as `unique`

[5] This visitor must be a post-order visitor, in order to correctly check expressions such as `Iterator i = this.seq.getIter()`. A post-order visitor 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 by the user with very little effort

[7] We used JavaCC [13] 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 manipulations in Java.

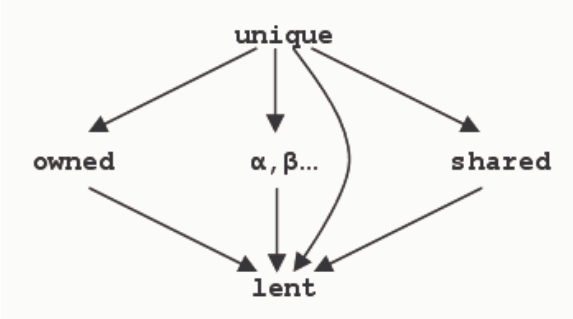

Fig. 7. Arrow means data can flow between variables with two annotations.

- 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 consists also 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
- ClassInstanceCreation: constructor rules
- MethodInvocation: method call rules
- ReturnStatement: assignment
- 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 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 aliasing examples available in the ArchJava distribution and converting them from AliasJava syntax to Java syntax, with JavaD annotations.

5.2 Courses Examples

We were also able to annotate a small Java program that corresponds to a three-tier course registration system. The tool identified several instances of unsupported coding patterns discussed later. We used the Eclipse refactoring ("Extract Local Variable") to replace them by supported coding patterns. In addition, we used the Eclipse Find/Replace feature to replace all instances of `java.lang.String` with `@Domain("shared")String`.

In addition, we replaced all uses of `java.util.ArrayList` by the annotated `Sequence` Abstract Data Type. This in turn required changes to use the Iterator constructor to iterate over the elements of the loop.

We found the JavaD tool helpful in pinpointing incorrect annotations. The most common annotation errors involved not passing the appropriate parameters, and forgetting to mark variables (used in annotations) as final. 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 problems in the tool.

[8] Currently, the tool does not support annotating library code. This is deferred for future work.

6 Limitations and Future Work

6.1 Missing Features.

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

- Add support for method parameters;
- Add back support for the `owner` annotation (the current support in the AliasJava compiler is broken); the original Sequence example shown earlier did make good use of it (See Figure 5). However, we rewrote it by passing the `Mowner` as an explicit domain parameter (the first one) (See Figure 6);
- Add different checks for public and private domains;
- Add additional sanity checks for the annotations; e.g., checks that the `@DomainInherits` does not reference to a nonexistent superclass or implemented interfaces; other needed checks include not allowing "owned", "lent" or "unique" as private domains in the `@Domains` annotation;
- Improve some of the error messages, Some of the error messages are quite cryptic, e.g., "Alias parameters [] of new Course(objCourseFile.readLine()) instance don't match [objectsDom] alias parameters of receiver at call ..." (for the "Before" construct in Figure 8). In this case, the developer has no indication that she must simply declare two local variables and add annotations to them;
- Add domain link specifications (`@DomainLinks` and `@DomainAssumes`) and the associated checks.

Some tool-specific future work might involve:

- Create an Eclipse builder so the analysis is invoked continuously as the program is being modified;
- Add 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 the bugs;
- Add the ability to turn off defaults; from a program comprehension point of view, 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

Java 1.5 annotations are too limiting. Some of their limitations include:

- A declaration cannot have multiple annotations for the same annotation type;
- Annotation types cannot have members of the their own type;
- Annotation types cannot extend any entity (class, interface or Annotation etc).
- 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")`.

These restrictions prevent 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 `@Domain(``owned <owned>'')`. This means that the developer using annotations is 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 lack of "positional" arguments prevented us from expressing constructs of the form `@Domains({"public1", "public2"}, private = {"private1", "private2"})`.

In addition, Java 1.5 annotations cannot be added to certain expressions. In particular, we found two kinds of expressions to be problematic, namely New Expressions (See Figure 8) and Cast Expressions (See Figure 9). These expressions will need to be replaced with equivalent constructs, namely by declaring a local variable and adding the appropriate annotations to it.

As an an alternative, we could have used stylized comments, but we think that comments would have been much less structured, and much easier to overlook by developers than annotations. Most editors, including the Eclipse 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);
}
```

Fig. 8. Re-writing a new expression by declaring a local variable 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";
    }
}
```

Fig. 9. Re-writing cast expression by declaring a temporary local variable.

6.3 Future Work

There are several directions that we can pursue next.

6.3.1 Support 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. A preferred approach is to not make changes to library or third-party code (which may not be available, and when it is, evolves separately), and to place annotations in separate files.

Even when storing annotations in separate files, there should be a way to also reduce the annotation burden, e.g., by using pattern matching to annotate all parameters or return types of type `java.lang.String` be annotated with `@Domain(‘‘shared’’)`.

6.3.2 Support 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" [9], "readonly" reference [14], among many possible ones.

6.3.3 Support 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 [10] 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 is that the inference algorithm can in no way break an existing program by inserting bad annotations.

6.3.4 Tolerate Local Inconsistencies.

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" [10]. Based on our own experience, we agree that it might be useful to have escape mechanisms to tolerate small local inconsistencies and to suppresses warnings on innocuous code. Another example is to allow, under certain conditions, to have a class override a method defined in an interface, and to change some of the aliasing annotations on the overridden methods (currently prohibited by AliasJava rules).

6.3.5 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 in case studies. 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 in [15]. 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.

We think this tool could also spur future activity, notably in the areas of extending the language with richer annotations and building interactive annotation inference tools.

References

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

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

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

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

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

[6] 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.

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

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

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

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

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

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

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

[14] 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.

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

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

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

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

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

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