Bringing Ownership Domains to Mainstream Java

Marwan Abi-Antoun
Carnegie Mellon University,
5000 Forbes Avenue,
Pittsburgh, PA 15213
marwan.abi-antoun@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University,
5000 Forbes Avenue,
Pittsburgh, PA 15213
jonathan.aldrich@cs.cmu.edu

Abstract

AliasJava is a type annotation system that extends Java to express how data is confined within, passed among, or shared between objects in a software system. We present an implementation of the AliasJava system as Java 1.5 annotations and an analysis using the Eclipse infrastructure.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Ownership domains, encapsulation, uniqueness, aliasing

1. Introduction

Aliasing in object-oriented programs can cause a failure of encapsulation and can be the source of many unintended side effects (See Figure 1). However, aliasing cannot be eliminated entirely from useful object-oriented programs.

2. AliasJava

AliasJava [5] implements the Ownership Domains system [4] as an extension of Java that involves mainly adding annotations to reference types. AliasJava divides objects into conceptual groups called ownership domains (where each object belongs to a single ownership domain) and allows developers to specify high-level policies that govern references between ownership domains and thus express controlled aliasing. AliasJava can express how data is confined within an object (`owned` annotation), passed linearly from one object to another (`unique` annotation), shared temporarily (`lent` annotation) within a method invocation or shared persistently and globally (`shared` annotation). 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 = ...`). Figure 2 shows a possible fix for the aliasing bug identified in Figure 1, an actual bug in an early version of the JDK.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

```
class JavaClass {
  private List signers;

  public List getSigners() {
      return this.signers;
  }
}
```

Figure 1. `Class.getSigners` returned the internal list rather than a copy, allowing malicious clients to pose as trusted code by modifying the `signers` object.

```
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 2. AliasJava version returning a copy of the list of signers in domain parameter `data`. The compiler will raise an error if the `owned signers` field is a returned by a public method.

3. Annotation Design

Unlike some of the other ownership type systems which are only paper designs, AliasJava has had a publicly available open-source compiler for a few years [1]. AliasJava was previously implemented as a non-backwards compatible extension to Java using a custom infrastructure. We re-implemented the AliasJava language and analysis as annotations using the annotation facility in Java 1.5 and the Eclipse Java Development Tooling (JDT) infrastructure.

Re-implementing the language using annotations improves the adoptability of the ownership domains technique by mainstream Java developers. First, this leads to improved tool support; in particular, we wanted all the capabilities of the Eclipse environment to become available to AliasJava programs. Second, using annotations makes it easier to extend the AliasJava language in a backwards-compatible way to support additional features such as external uniqueness [6] among others. Finally, using annotations allows developers to incrementally and partially specify annotations on large code bases while maintaining a running system.

Since our main goal 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

```
@Domains({"owned"}) // Same as default
@DomainParams({"data"})
class JavaClass {
  private @Domain("owned") List signers;

  public @Domain("data") List getSigners() {
   @Domain("data") List copy = new List();
   for(int i = 0; I < this.signers.size();i++)
      copy.add(this.signers.get(i));
   return copy;
  }
}
```

Figure 3. Using Java 1.5 annotations on example in Figure 2.

to make the annotations as usable as possible using the following strategies. First, the analysis only generates warnings about inconsistent annotations and displays them in the Eclipse Problems window. Second, we supply reasonable defaults to reduce the annotation burden. Finally, our approach involves purely a static analysis and non-executable annotations that do not interfere with the running of the program, unlike AliasJava where there may be some runtime exceptions related to checks for bad casts inserted by the AliasJava compiler. As a result, our annotation-based system is unsound at casts.

Our current implementation consists of the following Java 1.5 annotations which we illustrate with code snippets from the Signers example. Other annotations are being added to support various features such as local warning suppression. An example using these annotations is shown in Figure 3.

@Domain: Specify the actual annotation, the actual parameters and the actual array parameters on a local variable, field, method parameter and method return type.
Format: $annot < params, \ldots > [arrayParams, \ldots]$ where

- $annot$ can be any alias annotation, or refer to the public domain of an object, e.g., $seq.iters$;
- $< params, \ldots >$ (angle brackets) optional annotation for the ordered list of actual domain parameters;
- $[arrayParams, \ldots]$ optional annotation for the list of actual array parameters, by order of array dimension.

Examples:

```
private @Domain("owned") signers;
public void run(@Domain("lent[shared]")String args[]){
```

@Domains: declare domains on a type (class or interface).
Format: $name$
Examples:

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

@DomainParams: declare domain parameters on a type (class or interface).
Format: $name$
Examples:

```
@DomainParams({"data"})
class JavaClass
```

@DomainInherits: pass parameters to superclass or implemented interfaces
Format: $typename < parameter, \ldots >, \ldots$
Examples:

```
@DomainParams({"a"}) interface I { ...
```

```
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 4. Re-writing a new expression by declaring a few local variables with the appropriate annotations.

```
@DomainParams({"a","b"})
@DomainInherits({"I<a>"}) interface II extends I {...
```

Additional annotations specify the aliasing of method and constructor receivers and permissions between domains.

4. Limitations

Using annotation imposes several restrictions since Java 1.5 only allows annotations on variable declarations: there are cases where temporary variables need to be declared just for the purpose of specifying annotations such as for new expressions, cast expressions, and arguments for method and constructor invocation (see Figure 4).

5. Implementation

The analysis consists of several visitors on the Abstract Syntax Tree (AST) maintained by the Eclipse JDT. The implementation also uses a lightweight data flow analysis framework also implemented using the Eclipse JDT to check that unique data is passed linearly. Additional details and examples can be found in the companion technical report [2].

6. Conclusions

We presented a re-implementation of the AliasJava language and analysis as a set of Java 1.5 annotations, using the Eclipse infrastructure. Unlike many other ownership type systems, AliasJava has been evaluated on actual object-oriented programs (see case studies in [5] and [3]). This tool can encourage additional case studies to evaluate the true benefits of using ownership domains by annotating and evolving real object-oriented implementations.

References

[1] ArchJava website. http://www.archjava.org. 2006.

[2] M. Abi-Antoun and J. Aldrich. JavaD: Bringing ownership domains to mainstream java. Technical Report CMU-ISRI-06-110, Carnegie Mellon University, 2006.

[3] M. Abi-Antoun, J. Aldrich, and W. Coelho. A case study in re-engineering a legacy application to enforce architectural control flow and data sharing. *Journal of Systems and Software*, 2006. To appear.

[4] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.

[5] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, 2002.

[6] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.