Inferring AJ Types for Concurrent Libraries

Wei Huang Ana Milanova

Rensselaer Polytechnic Institute
Troy, NY, USA
{huangw5, milanova}@cs.rpi.edu

Abstract

Data-centric synchronization advocates data-based synchronization as opposed to control-based synchronization. It is more intuitive and can make correct concurrent programming easier. Dolby et al. [9] proposed AJ, a type system for data-centric synchronization, and showed that Java programs can be refactored into AJ. Unfortunately, programmers still have to add synchronization constructs manually (in the form of AJ type annotations), and the burden on programmers is high. In this paper we propose a type inference technique that infers AJ types for concurrent libraries. Our technique significantly reduces the amount of annotations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.1.5 [*Programming Techniques*]: Object-oriented Programming

General Terms Languages, Theory

1. Introduction

Data races and *atomicity violations* are difficult to prevent in a multi-threaded program. Traditional approaches use synchronization for ordering instructions in order to prevent data races. These approaches are control-centric, because programmers have to protect all *accesses* to shared memory locations. Control-centric approaches are error-prone and inflexible. First, shared memory locations are not easy to identify because of the presence of aliasing in object-oriented programming. In addition, it is also hard to control granularity of synchronization. When adding or removing memory locations to be synchronized, a programmer has to carefully reorganize the instruction sequences.

Data-centric synchronization is a technique which advocates data-based synchronization as opposed to control-based synchronization. In short, programmers specify an *atomic set* of semantically-related locations; these locations must be synchronized consistently. Dolby et al. [9] proposed AJ, a type system for data-centric synchronization. AJ provides a correctness guarantee called atomic-set serializability, which prevents data races and other concurrency errors. Dolby et al. [9] show that Java programs can be refactored into AJ. Unfortunately, programmers still need to do significant work to add synchronization constructs in the form of AJ type annotations, and the overhead is relatively high.

FOOL '12 October 22, 2012, Tucson, AZ, USA.


```
1  public class Counter {
2    int val;
3    int get() { return val; }
4    void dec { val−−; }
5    void inc { val++; }
6  }
7  Counter c = new Counter();
8  c.inc();
9  c.dec();
10 ...
```

Figure 1. A simple counter class example from [9]. We omit the atomicset a and atomic(a) annotations because we assume that every class has exactly one atomic set and all fields are protected by this atomic set.

We propose a technique that infers AJ types for concurrent libraries. Programmers specify a small number of AJ annotations to express the design decisions, from which our system would automatically infer the remaining ones and verify the inference result. Our approach reduces the annotation burden significantly. It requires 42 alias annotations for the 11826 LOC Java Collections library. In contrast, the approach from Dolby et al. [9] requires 370 alias annotations. Thus, our approach achieves almost 90% reduction.

2. Data-centric Synchronization with AJ

2.1 Overview

AJ [9] extends Java with annotations that support data-centric synchronization. The type system in our paper, called AJ-lite, differs from AJ [9] in two ways. First, AJ-lite assumes exactly one atomic set, named a, per class and all fields of the class are protected by this atomic set. The atomic set is retrieved by referencing a ghost field a, i.e. this.a. This is a simplification of AJ, but it is consistent with the implementation presented in [9]. Each atomic set has a logical lock protecting all fields of the object. The methods of the class are *units of work* which preserve the consistency of the atomic set. Second, AJ-lite replaces the *internal class* in AJ with *internal references*. In AJ, internal is an annotation on class declarations. AJ requires that every instance of an internal class is tracked by the type system, and not leaked outside of the object that constructed it. In contrast, in AJ-lite *internal* is an annotation on references; AJ-lite tracks *internal* references and disallows leaks outside of the object that constructed them. AJ-lite allows a class to have both internal and non-internal references. These two differences between AJ and AJ-lite do not violate the correctness property of AJ, i.e., the atomic set serializability guarantee. We justify this claim in Section 2.5.

Figure 1 shows a simple Counter class with atomic increment and decrement methods. Each Counter object has its own atomic set, protecting its only field val. Intuitively, when a thread accesses a Counter object, it must hold the logical lock associated with this atomic set. Thus, the accesses (increments and decrements of field val) are serialized and therefore consistent.

```
1  class PairCounter {
2    int diff;
3    A* Counter low = new A* Counter();
4    A* Counter high = new A* Counter();
5    void incHigh() {
6      high.inc();
7      diff = high.get()−low.get();
8    }
9      ...
10 }
```

(a) PairCounter with aliasing atomic sets.

```
1  class PairCounter {
2    int diff;
3    A! Counter low = new A! Counter();
4    A! Counter high = new A! Counter();
5    void incHigh() {
6      high.inc();
7      diff = high.get()−low.get();
8    }
9    ...
10 }
```

(b) PairCounter with internally aliasing atomic sets.

Figure 2. A pair counter class. The example is taken from [9].

In many cases, an atomic set must protect fields of more than one object. AJ supports merging atomic sets using *alias annotations*. Figure 2(a) shows a PairCounter class which has two integer counters and one method incHigh that updates the difference between counters. Merging is done by *aliasing* the atomic set of each Counter with the atomic set of the PairCounter object. AJ-lite uses and qualified type $\mathcal{A}*$ Counter. This corresponding to $|\mathsf{a} = \mathsf{this.a}|$ Counter in AJ, where a refers to the atomic set of the Counter object and this.a refers to the atomic set of the enclosing PairCounter object. Intuitively, this means that at runtime, a thread that accesses PairCounter, and/or one of the Counter objects, must hold the logical lock of PairCounter as well as the locks of the two Counter objects. Note that the locks are only logical — an actual implementation can choose to map each logical lock to a distinct physical lock, merge aliased logical locks into a single physical lock, and so on.

If the low and high counter objects remain confined within the PairCounter, that is, all accesses to these counter objects go through their enclosing PairCounter object, then the Counter objects do not need locks because they are protected by the lock of PairCounter. Thus, if the programmer knows (or an analysis proves) that the counter objects are never exposed, then he/she may annotate references low and high with the internal alias qualifier $\mathcal{A}!$. The typing of PairCounter will be as in Figure 2(b). This internal alias qualifier $\mathcal{A}!$ corresponds to the internal annotation on classes in AJ which we discussed earlier.

2.2 AJ-lite Qualifiers

We now formally introduce AJ-lite's type qualifiers. There are three source-level qualifiers in AJ-lite, i.e., the universal set of qualifiers $U_{\mathsf{AJ\text{-}lite}} = \{\mathcal{A}*, \mathcal{A}?, \mathcal{A}!\}$:

- $\mathcal{A}*$: The atomic set of the object referenced by an $\mathcal{A}*$ reference x is aliased with the atomic set of the current (i.e., this) object, i.e. x.a = this.a. $\mathcal{A}*$ corresponds to the $|\mathsf{a} = \mathsf{this.a}|$ annotation in AJ.
- $\mathcal{A}?$: The atomic set of the object that is referenced by an $\mathcal{A}?$ reference x may or may not be aliased to the set of the current object. In other words, we do not know whether this.a and x.a are aliased or not. $\mathcal{A}?$ corresponds to the implicit default annotation in AJ.
- $\mathcal{A}!$: The atomic set of an $\mathcal{A}!$ object is aliased to the current object. $\mathcal{A}!$ is different from $\mathcal{A}*$ because it forbids exposure of the object outside of the object that constructed it, i.e. the $\mathcal{A}!$ object is internal to the current object. In contrast, an $\mathcal{A}*$ object can be accessed by arbitrary objects. $\mathcal{A}!$ corresponds to the internal annotation on classes in AJ.

The qualifiers form the following subtyping hierarchy:

$$\mathcal{A}* <: \mathcal{A}?$$

Therefore, $\mathcal{A}*$ references can be assigned to $\mathcal{A}?$ ones. However, $\mathcal{A}?$ ones cannot be assigned to $\mathcal{A}*$ ones. This is consistent with AJ, which allows dropping the alias annotation but disallows adding an alias annotation. The difference between $\mathcal{A}!$ and $\mathcal{A}*$ is that $\mathcal{A}!$ is not a subtype of $\mathcal{A}?$ and therefore, $\mathcal{A}!$ references cannot be assigned to anything but $\mathcal{A}!$ references.

2.3 Viewpoint Adaptation

In AJ and AJ-lite, viewpoint adaptation is used when deciding the types of fields at field access, and the types of formal parameters and method returns at method call. Consider the field access x.f where both x and f are declared as $\mathcal{A}*$. x being $\mathcal{A}*$ means that the x object and the current object have their atomic sets aliased. Similarly, the field f being $\mathcal{A}*$ means that the x object and the f object have their atomic sets aliased as well. Therefore, we can conclude that the type of x.f is $\mathcal{A}*$ as well. Consider another field access y.g where y is $\mathcal{A}?$ and g is $\mathcal{A}*$. Because we cannot decide whether this's atomic set is aliased to g's atomic set, y.g is of type $\mathcal{A}?$. Therefore, the types of fields at field access and the types of formal parameters and method returns at method call, depend on not only their declared types, but also the type of the *receiver*, which represents the access context.

Both AJ and AJ-lite encode this by means of *viewpoint adaptation*. Viewpoint adaptation is a concept from Universe Types [5, 7, 8], which can be adapted to Ownership Types [4]. Viewpoint adaptation of a type q from the point of view of another type q', results in the adapted type q''. This is written as $q' \triangleright q = q''$. Viewpoint adaptation in AJ-lite is defined as follows (Undefined adaptations would be considered as type errors):

$$\begin{array}{lcll} \mathcal{A}! & \triangleright & q & = & q \\ \mathcal{A}* & \triangleright & q & = & q \\ \mathcal{A}? & \triangleright & \mathcal{A}* & = & \mathcal{A}? \\ \mathcal{A}? & \triangleright & \mathcal{A}? & = & \mathcal{A}? \end{array}$$

The first two rules state that adapting any type q from the point of view of $\mathcal{A}!$ or $\mathcal{A}*$ results in q. Recall the field access x.f where both x and f are $\mathcal{A}*$. We can decide the type x.f by using viewpoint adaptation: $\mathcal{A}* \triangleright \mathcal{A}* = \mathcal{A}*$. The last two adaptation rules state that adapting q ($q \neq \mathcal{A}!$) from the point of view of $\mathcal{A}?$ results in $\mathcal{A}?$. Recall the other field access y.g where y is $\mathcal{A}?$ and g is $\mathcal{A}*$. We can decide the type of y.f is $\mathcal{A}? \triangleright \mathcal{A}* = \mathcal{A}?$. The reason that the rules forbid adapting $\mathcal{A}!$ from the point of view of $\mathcal{A}?$ is to guarantee that internal references would never escape to unknown context.

The above rules are consistent with the rules from Dolby et al. [9]:

$$\begin{array}{lcl} adapt(\mathsf{C}, t) & = & \mathsf{C} \\ adapt(\mathsf{C}|a = \mathsf{this}.b|, \mathsf{D}|b = \mathsf{this}.c|) & = & \mathsf{C}|a = \mathsf{this}.c| \end{array}$$

where $adapt(t, t')$ is the view of type t from the point of view of type t'. Here the first rule expresses that an $\mathcal{A}?$ type C adapted from any point of view, results in an $\mathcal{A}?$ type, as it is in our rules. The second rule states that if the adaptee type t is aliased (i.e., we have $\mathsf{C}|a = \mathsf{this}.b|$) then the adapter type t' must be aliased as well ($\mathsf{D}|b = \mathsf{this}.c|$), and the result of the adaptation is an aliased type

$$
\begin{array}{lll}
cd & ::= \mathsf{class\ C\ extends\ D\ \{}\ \overline{fd}\ \ \overline{md}\ \mathsf{\}} & \textit{class} \\
fd & ::= t\ \mathsf{f} & \textit{field} \\
md & ::= t\ \mathsf{m}(t\ \mathsf{x})\ q\ \mathsf{\{}\ \overline{t\ \mathsf{y}}\ \mathsf{s;\ return\ y\ \}} & \textit{method} \\
\mathsf{s} & ::= \mathsf{s;s} \mid \mathsf{x} = \mathsf{new}\ t() \mid \mathsf{x} = \mathsf{y} & \textit{statement} \\
 & \quad \mid \mathsf{x.f} = \mathsf{y} \mid \mathsf{x} = \mathsf{y.f} \mid \mathsf{x} = \mathsf{y.m(z)} & \\
t & ::= q\ \mathsf{C} & \textit{qualified type} \\
q & ::= \mathcal{A}* \mid \mathcal{A}? \mid \mathcal{A}! & \textit{qualifier}
\end{array}
$$

Figure 3. Syntax of a core OO language. C and D are class names, f is a field name, m is a method name, x, y and z are names of local variables and formal parameters, including implicit parameter this, and qualifier q is independent of the Java type. Qualifier q at the method declaration qualifies implicit parameter this.

$\mathsf{C}|a = \mathsf{this}.c|$. This seems different from AJ-lite because AJ-lite allows adapting $\mathcal{A}*$ from the point of view of $\mathcal{A}?$, but they are essentially the same. Remember that AJ allows dropping the alias annotation. Therefore, adapting $\mathcal{A}*$ from the point of view of $\mathcal{A}?$ in AJ-lite is essentially the same as dropping the alias annotation of $\mathsf{C}|a = \mathsf{this}.b|$ and apply the first adaptation rule of AJ, and the result is consistent — it is $\mathcal{A}?$ in AJ-lite and C in AJ.

2.4 Typing Rules

Syntax For brevity, we restrict our formal attention to a core calculus in the style of Dolby et al. [9] whose syntax appears in Figure 3. The language models Java with a syntax in a "named form", where the results of field accesses, method calls, and instantiations are immediately stored in a variable. Without loss of generality, we assume that methods have parameter this, and exactly one other formal parameter. Features not strictly necessary are omitted from the formalism, but they are handled correctly in the implementation. We write $\overline{t\ \mathsf{y}}$ for a sequence of local variable declarations.

In contrast to a formalization of standard Java, a type t has two orthogonal components: type qualifier q (which expresses the alias annotation) and Java class type C. The AJ-lite type system is *orthogonal* to (i.e., independent of) the Java type system, which allows us to specify typing rules over type qualifiers q alone.

Typing rules The typing rules are shown in Figure 4. These rules are generic and enforce standard subtyping constraints with viewpoint adaptation at field access and method call. For example, at field write (TWRITE), f's type is adapted from the point of view of x; the resulting adapted type must be a supertype of the type of the right-hand-side y of the assignment. The generic rules are part of an inference and checking framework for ownership-like type systems [14]. The framework takes as input (1) the universe of type qualifiers, in our case $U_{\mathsf{AJ\text{-}lite}} = \{\mathcal{A}!, \mathcal{A}*, \mathcal{A}?\}$, (2) the subtyping hierarchy of type qualifiers, in our case $\mathcal{A}* <: \mathcal{A}?$, (3) the viewpoint adaptation function, in our case, as it was specified earlier (Section 2.3), and (4) the additional $\mathcal{B}$ constraints, which are constraints imposed by individual type systems.

In AJ no rules demand additional constraints. Therefore, all $\mathcal{B}$ sets are empty. We elaborate on the rule for (TCALL). Note that constraint $q_{\mathsf{y}} <: q_{\mathsf{y}} \rhd q_{\mathsf{this}}$ prevents a leak of an $\mathcal{A}!$, i.e., internally aliased reference, which is supposed to be encapsulated by its enclosing object. We also note here, that implicit parameters this can only be $\mathcal{A}!$ or $\mathcal{A}*$. Implicit parameter this is always internally aliased or aliased as a result of our decision that every class has an atomic set. Intuitively, this is $\mathcal{A}*$, except when the method is called on an internal receiver, in which case it must be $\mathcal{A}!$ in order to prevent a leak.

The above mentioned constraint enforces the notion of the internal class from [9]. Figure 5 shows an example. The return type of m in class C is $\mathcal{A}?$— m is public and can be invoked at arbitrary points. As a result, the return type of id and subsequently

$$
\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TNEW})}(q_{\mathsf{x}}, q)}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}\ (\text{TNEW})
$$

$$
\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TASSIGN})}(q_{\mathsf{x}}, q_{\mathsf{y}})}{\Gamma \vdash \mathsf{x} = \mathsf{y}}\ (\text{TASSIGN})
$$

$$
\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \rhd q_{\mathsf{f}} \quad \mathcal{B}_{(\text{TWRITE})}(q_{\mathsf{x}}, q_{\mathsf{f}}, q_{\mathsf{y}})}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}\ (\text{TWRITE})
$$

$$
\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad q_{\mathsf{y}} \rhd q_{\mathsf{f}} <: q_{\mathsf{x}} \quad \mathcal{B}_{(\text{TREAD})}(q_{\mathsf{y}}, q_{\mathsf{f}}, q_{\mathsf{x}})}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}\ (\text{TREAD})
$$

$$
\frac{\begin{array}{c}\mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q \to q' \quad \Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \\ q_{\mathsf{z}} <: q_{\mathsf{y}} \rhd q \quad q_{\mathsf{y}} \rhd q' <: q_{\mathsf{x}} \quad q_{\mathsf{y}} <: q_{\mathsf{y}} \rhd q_{\mathsf{this}} \\ \mathcal{B}_{(\text{TCALL})}(\mathsf{m}, q_{\mathsf{y}}, q_{\mathsf{x}})\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}\ (\text{TCALL})
$$

Figure 4. Generic typing rules. The rules enforce standard subtyping constraints as well as additional constraints $\mathcal{B}$ that can be imposed by a concrete type system.

```
1  public class Id {
2    Id id() {
3      Id x = this;
4      return x;
5    }
6  }
```

```
7  class C {
8    public Id m() {
9      A! Id y; Id z;
10     y = new A! Id();
11     z = y.id();
12     return z;
13   }
14 }
```

Figure 5. A leak of implicit parameter this. Example from [9].

x and this in id cannot be $\mathcal{A}!$. Suppose that the return of id and x are $\mathcal{A}?$, and this of id is $\mathcal{A}*$; thus, id type checks. The constraint $q_{\mathsf{y}} <: q_{\mathsf{y}} \rhd q_{\mathsf{this}}$ causes type checking at call $\mathsf{z} = \mathsf{y.id()}$ to fail: we have $\mathcal{A}!$ y but an $\mathcal{A}*$ this, and $\mathcal{A}? \rhd \mathcal{A}!$ is undefined. This is the desired behavior because it disallows the leak of the internal Id object.

Arrays are handled by specifying two types, one for the array element and one for the array object. For example, following Java 8 syntax [10]

$$\mathcal{A}?\ \mathsf{Object}\ \mathcal{A}!\ []\ \mathsf{signers};$$

declares an $\mathcal{A}!$ array signers which stores $\mathcal{A}?$ elements of type Object. The type of the array object in the above declaration is from the point of view of the declaring class, while the type of the element object is from the point of view of the array object. Given these two types (if the programmer chooses to specify these types), there exists a unique array field type. The array field type gives the type of the element from the point of view of the array object. In the above example, the array field type is $\mathcal{A}?$. Thus, array accesses are checked as field accesses: For example, the statement $\mathsf{s} = \mathsf{signers[i]}$; generates the following constraint:

$$q_{\mathsf{sig}} \rhd q_{[]} <: q_{\mathsf{s}}$$

```
1  class Transfer {
2    void transfer(unitfor Counter from, unitfor Counter to) {
3      from.dec();
4      to.inc();
5    }
6  }
```

Figure 6. Adding atomic sets to a unit of work.

where q_{s} gives the type of s, q_{sig} gives the type of the signers array object, and $q_{[]}$ gives the array field type of that array object.

AJ [9] provides an additional annotation, unitfor, which is used to annotate formal parameters in bulk methods. unitfor unions the atomic set of the actual argument with the atomic set of the receiver of the method for the duration of the execution of the method. Consider the example in Figure 6. The from and to counters must be updated atomically. The unitfor construct ensures that atomic sets of the from and to objects are merged with the atomic set of the receiver of the transfer method which ensures the consistency of the update.

unitfor is a dynamic construct. It has no effect on the static type systems (it does not appear in the static typing rules for AJ in [9]). unitfor cannot be easily inferred because the consistency requirements of bulk methods are highly dependent on program semantics as it is in the Transfer example. In this paper, we assume that unitfor annotations are provided by the programmer and focus our inference effort on the alias annotations.

2.5 Correctness Argument

As discussed in Section 2.1, AJ-lite differs from AJ in two ways. We make an informal argument that these two differences do not violate the correctness property of AJ, namely, atomic set serializability.

The first difference is that AJ-lite has exactly one atomic set per class and all fields in the class are protected by this atomic set, while AJ allows more than one atomic set and some fields can be excluded from any atomic sets. In fact, AJ-lite captures a special case of AJ, and it is also consistent with AJ's current implementation [9]. Therefore, AJ-lite's simplification of AJ still has the correctness guarantee as proved in [9]. However, this simplification may hinder concurrent accesses to data structures designed for sharing. We will address this restriction in future work.

The second difference is that AJ-lite uses *internal references* instead of the *internal class* in AJ. The adaptation rules of AJ-lite enforce that an object referenced by an $\mathcal{A}!$ variable either (1) remains protected by the atomic set of its creating object, or (2) escapes to an object whose atomic set is aliased with its creating object. (1) is straightforward when an $\mathcal{A}!$ object is never exposed to the outside (this is exactly the same as ownership encapsulation [4]). (2) is enforced by the first two adaptation rules $\mathcal{A}!\triangleright q = q$ and $\mathcal{A}*\triangleright q = q$. An $\mathcal{A}!$ variable remains $\mathcal{A}!$ when it is adapted from the viewpoint of $\mathcal{A}!$ or $\mathcal{A}*$. For example, y.f where y is $\mathcal{A}*$ and f is $\mathcal{A}!$ is of type $\mathcal{A}!$. Because y's atomic set is aliased with the current this's atomic set, thus the $\mathcal{A}!$ f escapes to this and is protected by this's atomic set. Also, the adaptation of $\mathcal{A}!$ from the viewpoint of $\mathcal{A}?$ is not allowed, thus an $\mathcal{A}!$ variable will not escape to an $\mathcal{A}?$ context. Because of (1) and (2), we know that an $\mathcal{A}!$ variable is always protected by the atomic set of its enclosing data structure and it behaves the same as an $\mathcal{A}*$ variable. AJ-lite's extension on internal reference does not violate the atomic-set serializability property proved by [9]. Further, $\mathcal{A}!$ reference provides other optimization opportunities in the implementation. For example, if a class is always annotated $\mathcal{A}!$, we can conclude that this class can safely get rid of its atomic set, which could improve the performance of the program.

2.6 LinkedList Example

Figure 7 shows a LinkedList example with alias annotations as inferred by our analysis. Note that in general, AJ-lite would require at least some programmer annotations. Programmer annotations will denote semantically related objects that must belong to the same atomic set. Conversely, annotations could denote unrelated objects which must have separate atomic sets in order to increase parallelism. For example, the LinkedList and the ListItr objects are semantically related and must be aliased: a modification to the LinkedList while iteration is in progress will result in incorrect behavior of the iterator (e.g. we may get a ConcurrentModificationException in Java). Conversely, a new array created in the toArray method of a collection, is unrelated to the collection object; the programmer can annotate the array creation site as $\mathcal{A}?$.

The AJ example in Figure 7 requires no manual annotations. The ListItr object is inferred as $\mathcal{A}*$ (clearly, the LinkedList and its iterator are related, and must belong to the same atomic set). The Entry objects are inferred as $\mathcal{A}!$ because they are accessed only from the LinkedList and ListItr objects which belong to the same atomic set (thus, the Entry objects are internal to this atomic set).

3. Type Inference and Checking

Our type inference is phrased in the general framework for specification, inference and checking of ownership-like type systems [14]. Recall that the framework takes as input (1) the universe of type qualifiers, in our case $U_{\mathsf{AJ\text{-}lite}} = \{\mathcal{A}!, \mathcal{A}*, \mathcal{A}?\}$, (2) the subtyping hierarchy of type qualifiers, in our case $\mathcal{A}* <: \mathcal{A}?$, (3) the viewpoint adaptation function, in our case, as it was specified earlier (Section 2.3), and (4) the additional $\mathcal{B}$ constraints, which are empty as argued earlier (Section 2.4).

Note that as with ownership type systems (e.g., Universe Types and Ownership Types) AJ-lite permits multiple valid typings. For example, all variables[1] in the program could be typed as $\mathcal{A}?$; the program will type check but will be unsafe in a sense that it will allow atomicity violations. Note that the correctness guarantee of AJ and AJ-lite, i.e., the atomic set serializability property, is with respect to the provided alias annotations. For example, if the programmer has missed to annotate as aliased objects LinkedList and ListItr in Figure 7, the program will type check, but it may throw an exception. Similarly, all variables can be typed $\mathcal{A}*$ which, again will type check, but will lose concurrency (as all objects will form one giant atomic set and the program will degenerate into a sequential program). Recall that we simplify the original AJ by assuming that every class has an atomic set and all fields belong to this atomic set.

In addition to the above parameters, which define the AJ-lite type system, the inference framework takes an additional parameter: an ordering of the qualifiers. The ordering expresses preference for typing of variables. The AJ-lite qualifiers are ordered $\mathcal{A}! > \mathcal{A}* > \mathcal{A}?$. This means (roughly) that if possible, a variable should be typed as $\mathcal{A}!$; in other words we prefer internally aliased. Otherwise (i.e., if $\mathcal{A}!$ is impossible), it should be typed as $\mathcal{A}*$. If neither $\mathcal{A}!$ or $\mathcal{A}*$ are possible, it should be typed as $\mathcal{A}?$. This ordering over qualifiers gives rise to an ordering over all valid typings: for two typings T_1, T_2, we have $T_1 > T_2$ iff T_1 types more variables as $\mathcal{A}!$ than T_2 or T_1 and T_2 type the same number of variables as $\mathcal{A}!$ but T_1 types more variables as $\mathcal{A}*$ than T_2. The highest ranked typing in this ordering is what we call the "best", or most desirable typing. Our goal is to infer the "best" typing.

The inference initializes all annotated variables to the singleton set that contains the programmer-provided annotation, default initial

[1] Term "variable" is used to refer to (1) allocation sites, (2) local variables, including formal parameters, (3) fields, and (4) method returns

```
1  public abstract class AbsList {
2    int size;
3    public int size() A* {
4      return size;
5    }
6    public abstract A* ListIterator iterator() A*;
7    public abstract void add(Object o) A*;
8    public abstract boolean addAll(AbsList c) A*;
9    public abstract Object get(int i) A*;
10 }

12 class Entry {
13   Object elem;
14   A! Entry next;
15   A! Entry prev;
16   Entry(Object elem, A! Entry next,
17       A! Entry prev) A! {
18     this.elem = elem;
19     this.next = next;
20     this.prev = prev;
21   }
22 }

23 class LinkedList extends AbstractList {
24   A! Entry header;
25   public LinkedList() A* {
26     header = new A! Entry(null,null,null);
27     header.prev = header;
28     header.next = header;
29   }
30   public void add(Object o) A* {
31     A! Entry p = header.prev;
32     A! Entry newEntry = new A! Entry(o,header,p);
33     header.prev = newEntry;
34     p.next = newEntry;
35     size++;
36   }
37   public A* ListIterator iterator() A* {
38     ListIterator it = new A* ListItr(this,header);
39     return it;
40   }
41 }

43 class ListItr implements ListIterator {
44   final A* LinkedList list;
45   private A! Entry header;
46   ListLtr(A* LinkedList l, A! Entry h) A* {
47     list = l;
48     header = h;
49   }
50   ...
51 }
```

Figure 7. The LinkedList example with annotations as inferred by our analysis. Unannotated variables are $\mathcal{A}?$. The annotation after method declaration gives the type of implicit parameter this.

type assignments for special variables (discussed below), and all remaining variables to $U_{\text{AJ-lite}}$, the universal set of AJ-lite qualifiers. Then it repeatedly examines each program statement and applies the statement's typing rule on the current set of qualifiers; it removes infeasible qualifiers from the sets until it reaches a fixpoint.

As an example, suppose that the iteration examines x = y where x is mapping to $U_{\text{AJ-lite}} = \{\mathcal{A}!, \mathcal{A}*, \mathcal{A}?\}$ and y is mapping to a singleton set $\{\mathcal{A}?\}$. When the inference examines the statement, it removes qualifiers $\mathcal{A}!$ and $\mathcal{A}*$ from the set for x, because neither $\mathcal{A}? <: \mathcal{A}!$ or $\mathcal{A}? <: \mathcal{A}*$ holds (which the typing rule for x = y requires as shown in Figure 4). The final result of the inference is a mapping from variables to *sets* of qualifiers. We derive a mapping from variables to AJ-lite types by mapping each variable to the maximal qualifier in its set. For AJ-lite, it is guaranteed that (1) this is a valid typing. This can be proven using a case-by-case analysis. In addition, it is verified by an independent type checker, which is part of our implementation. (2) this is the unique "best" typing according to the ordering on typings described in the previous paragraph. The proof of a general case, from which the above statement derives, is given in [14].

Our inference analysis uses programmer-provided annotations, default initial type assignments, and the above qualifier ordering in order to infer a desirable typing. Below, we describe the inference process.

Recall that in this work, we focus on the typing of concurrent libraries. We start from the following default initial type assignments:

1. All non-this parameters of *public* methods receive default type $\{\mathcal{A}?\}$. It is expected that, in general, arguments will be unrelated to the current object and they will maintain their own atomic set. Note that default annotations take place only if there is no programmer-provided annotation; if it is needed that certain parameters are aliased, the programmer can annotate those parameter, and the programmer-provided annotation would take precedence over the default.
2. All this parameters receive default $\{\mathcal{A}!, \mathcal{A}*\}$.
3. All return types of *public* methods receive default $\{\mathcal{A}*, \mathcal{A}?\}$. These methods can be called and return at arbitrary points.

All remaining variables are initialized to $\{\mathcal{A}!, \mathcal{A}*, \mathcal{A}?\}$. Note that if a variable has a programmer-provided annotation, that annotation overrides the default.

The programmer must examine the allocation sites in the library and annotate as many allocation sites as $\mathcal{A}?$ as possible. Some of the allocation sites are semantically related to the creating this object and must be aliased; others are unrelated and can have an independent atomic set. The analysis prefers $\mathcal{A}*$ over $\mathcal{A}?$ (as we discussed earlier). Thus, all unannotated allocation sites will be typed $\mathcal{A}!$ or $\mathcal{A}*$; more precisely, if a site cannot be typed $\mathcal{A}!$, then it will be typed $\mathcal{A}*$. Therefore, in order to increase parallelism, the programmer is encouraged to annotate as many allocation sites as $\mathcal{A}?$ as the semantics of the program permits.

In addition to annotating allocation sites, the programmer may choose to annotate parameters in order to override the above-mentioned defaults. For example, in the commonly-used idiom new X(this), the this object and the new X object are typically semantically related. However, if X's constructor is public, the formal parameter to which this is assigned to will be typed as $\mathcal{A}?$ by default. We would like to type this formal parameter as $\mathcal{A}*$ because this will allow us to identify more $\mathcal{A}!$ objects.

Next, we set the order over qualifiers as $\mathcal{A}! > \mathcal{A}* > \mathcal{A}?$ and run the inference analysis. The optimality property holds for this system and ordering, and the inference produces the best AJ typing. We prefer $\mathcal{A}*$ over $\mathcal{A}?$ because it presents optimization opportunities.

4. Experiments

We have typed a subset of the Java Collections library: LinkedList, ArrayList, HashMap and all related classes. The subset amounts to 11826 LOC and includes 63 files.

We use the defaults outlined above. We manually added 3 $\mathcal{A}$? annotations: one at the allocation site Object[] result = new Object $\mathcal{A}$? [size()] in toArray of AbstractCollection, one at Object[] result = new Object $\mathcal{A}$? [size] in toArray of LinkedList, and one at Object[] result = new Object $\mathcal{A}$? [size] in toArray of ArrayList. The newly created arrays can exist independently of their creating collection. In addition, we added $\mathcal{A}*$ annotations to parameters as follows: for every allocation site new X(...,this,...) where the constructor X was public, we annotated as $\mathcal{A}*$ the formal parameter to which this is assigned. For example, method iterator in class AbstractList contains statement return (Iterator) new AbstractList_Itr(this). The this object and the newly created iterator objects must be aliased. However, the constructor AbstractList_Itr is public, and if not annotated, its formal parameter will be typed as $\mathcal{A}$? by default. Therefore, we insert a manual annotation $\mathcal{A}*$ for its formal parameter. In addition, we provide several more manual annotations in order to override the public default: e.g., parameter n of public void setNext($\mathcal{A}*$ HashMap_Entry n) { ... } must be annotated as $\mathcal{A}*$; if not annotated, n receives default type $\mathcal{A}$? which forces all HashMap_Entry objects to be $\mathcal{A}$?. However, HashMap_Entry is semantically related to AbstractList_Itr and must be part of its atomic set (i.e., must be aliased). In total we added **32** $\mathcal{A}$? and $\mathcal{A}*$ annotations.

We compared the results of our inference with the manually annotated Collections library used to report results in [9]. [2] The majority of annotations were as in the manually annotated code. We observed several differences in AbsractMap. For example, the manually annotated code in [9] contains the following (we use our simplified annotations):

```
1  public boolean containsValue(Object value) {
2    A* Iterator i = entrySet().iterator();
3    if (value == null) {
4      while (i.hasNext()) {
5        A* Map_Entry e = (A* Map_Entry)i.next();
6        if (e.getValue() == null)
7            return true;
8      }
9    } else {
10     while (i.hasNext()) {
11       A* Map_Entry e = (A* Map_Entry)i.next();
12       if (value.equals(e.getValue()))
13           return true;
14     }
15   }
16   return false;
17 }
```

Our inference types the e at lines 5 and 11 as $\mathcal{A}$?. This is because we do not add a cast at the right-hand-side of the assignment. The return of public Object next() is $\mathcal{A}$? (the object it returns is not part of the container's internal structure), and the $\mathcal{A}$? annotation propagates to the e's. The iterator i at line 2 is inferred as $\mathcal{A}*$, just as in the manually annotated code.

In order to get exactly the same set of alias annotations as the manually annotated Collection library we would need **10** downcast $\mathcal{A}*$ annotations (it is a design decision to not add these downcasts). Thus, in total, we will have **42** alias annotations. Compared with [9], **42** vs. **370** is almost 90% reduction. As mentioned earlier, we do not infer unitfor annotations. Those annotations are difficult to infer as it would require dynamic analysis. In order to achieve consistency we would need to manually add about 53 unitfor annotations, as in [9].

Our implementation is part of the inference and checking framework described in [14]. The code for the framework is publicly available at `http://code.google.com/p/type-inference/`, including source.

[2] We obtained the annotated library from Prof. Jan Vitek.

5. Related Work

We briefly discuss related work on type systems for preventing data races and atomicity violations, and inference of pluggable types.

Abadi et al. [1] present a static race detection analysis for Java. The analysis is based on a type system that captures synchronization patterns. By checking programmer provided type annotations, the type system can guarantee the absences of data races if the synchronization and the type annotations are consistent. They also provide an inference algorithm to compute annotations automatically and a user interface to facilitate inspecting warnings generated by the checker. Abadi et al.'s inference algorithm and the inference algorithm used by Tip et al. [12, 18] are similar to our inference algorithm of AJ. These algorithms start with sets containing all possible answers and iteratively remove elements that are inconsistent with the typing rules. Our algorithm also uses a preference ranking over qualifiers to pick up the "best" typing for AJ.

Flanagan and Qadeer [11] present a type system for specifying and verifying the atomicity of methods for Java. The type system can check that the instructions of an *atomic* method are not interleaved with instructions from other threads for any arbitrary executions. They also implement an atomic type checker for Java and discover a number of atomicity violations in java.lang.String and java.lang.StringBuffer.

There are a number of works for inferring user-defined type qualifiers to reduce programmer's burden on annotations. Greenfieldboyce and Foster [13] present a framework called JQual for inferring user-defined type qualifiers in Java. JQual is effective for *source-sink* type systems, for which programmers need to add annotations to the sources and sinks and JQual infers the intermediate annotations for the rest of the program. Chin et al. [3] propose CLARITY for the inference of user-defined qualifiers for C programs based on user-defined rules, which can also be inferred given user-defined invariants. CLARITY infers several type qualifiers, including pos and neg for integers, nonnull for pointers, and tainted and untainted for strings.

There are also lots of works on inference of ownership types. Aldrich et al. [2] present an ownership type system and a type inference algorithm. Their inference creates equality, component and instantiation constraints and solves these constraints. Ma and Foster [16] propose Uno, a static analysis for automatically inferring ownership, uniqueness, and other aliasing and encapsulation properties in Java. Dietl et al. [6] present a tunable static inference for Generic Universe Types (GUT). Constraints of GUT are encoded as a boolean satisfiability problem, which is solved by a weighted Max-SAT solver. Milanova and Vitek [17] present a static dominance inference analysis, based on which they perform ownership type inference.

This work is closely related to our previous work on inference of ownership types [14], and reference immutability types [15]. All type systems are applications in our inference and checking framework.

6. Conclusions and Future Work

We presented an inference technique which infers AJ types for concurrent libraries. The technique reduces the number of alias annotations by approximately 90%. This result shows that our technique is feasible.

In the future we will expand our technique to infer types for whole programs in addition to libraries. Whole programs are harder than libraries because there is no easy way to assign defaults, the way we assign defaults in libraries. We will exploit opportunities for optimization due to internal aliasing and read-only. Our experience with ownership types suggests that there are many internally aliased objects and thus, significant opportunities for optimization.

In addition, our assumption that all fields belong to the single atomic set of a class may hinder concurrent access to data structures designed for sharing, e.g. immutable objects. We will improve the type system to address this restriction.

7. Acknowledgements

We thank Dr. Frank Tip, Dr. Mandana Vaziri and the anonymous FOOL reviewers for their detailed and extremely valuable comments on earlier versions of this paper.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.

[3] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, pages 264–278, 2006.

[4] D. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[5] D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers. Universe types for topology and encapsulation. In *FMCO*, pages 72–112, 2008.

[6] W. Dietl, M. D. Ernst, and P. Müller. Tunable static inference for generic universe types. In *ECOOP*, pages 333–357, 2011.

[7] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[8] W. Dietl and P. Müller. Runtime universe type inference. In *IWACO*, pages 72–80, 2007.

[9] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):1–48, Apr. 2012.

[10] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, 2012.

[11] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, number 5, pages 338–349, 2003.

[12] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, 2005.

[13] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *OOPSLA*, pages 321–336, 2007.

[14] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.

[15] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, 2012.

[16] K.-K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for java. In *OOPSLA*, pages 423–440, 2007.

[17] A. Milanova and J. Vitek. Static dominance inference. In *TOOLS*, pages 211–227, 2011.

[18] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. D. Sutter. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems*, 33(3):1–47, Apr. 2011.