A Data-Centric Approach to Synchronization†

JULIAN DOLBY, IBM T.J. Watson Research Center
CHRISTIAN HAMMER, Purdue University
DANIEL MARINO, UCLA
FRANK TIP, IBM T.J. Watson Research Center
MANDANA VAZIRI, IBM T.J. Watson Research Center
JAN VITEK, Purdue University

Concurrency-related errors such as data races are frustratingly difficult to track down and eliminate in large object-oriented programs. Traditional approaches to preventing data races rely on protecting instruction sequences with synchronization operations. Such control-centric approaches are inherently brittle as the burden is on the programmer to ensure that all concurrently accessed memory locations are consistently protected. Data-centric synchronization is an alternative approach that offloads some of the work on the language implementation. Data-centric synchronization groups fields of objects into *atomic sets* to indicate that these fields always must be updated atomically. Each atomic set has associated *units of work*, code fragments that preserve the consistency of that atomic set. Synchronization operations are added automatically by the compiler. We present an extension to the Java programming language that integrates annotations for data-centric concurrency control. The resulting language, called AJ, relies on a type system that enables separate compilation and supports atomic sets that span multiple objects and that also supports full encapsulation for more efficient code generation. We evaluate our proposal by refactoring classes from standard libraries as well as a number of multi-threaded benchmarks to use atomic sets. Our results suggest that data-centric synchronization is easy to use, and enjoys low annotation overhead, while successfully preventing data races. Moreover, experiments on the SPECjbb benchmark suggest that acceptable performance can be achieved with a modest amount of tuning.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming-parallel programming; D.2.4 [**Software Engineering**]: Software/Program Verification-reliability; D.3.3 [**Programming Languages**]: Language Constructs and Features — Data types and structures; F.3.1 [**Logics And Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Theory

Additional Key Words and Phrases: Concurrent Object-Oriented Programming, Data Races, Serializability, Programming Model

1. INTRODUCTION

Writing correctly synchronized concurrent programs is challenging. Whenever two threads access the same memory location there is the potential for a *data race* and for inconsistent results. Traditional techniques for concurrent programming have an operational, control-centric, flavor. Programmers must ensure that any access to a shared data location is protected by synchronized blocks or other system-specific concurrency

Authors' addresses: J. Dolby, F. Tip, M. Vaziri: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; C. Hammer, J. Vitek: Dept. of Computer Sciences, Purdue University, 305 N University Street, West Lafayette, IN 47907, USA; D. Marino: UCLA Computer Science Department, 3440 Boelter Hall, Los Angeles, CA 90095.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

control primitives. The challenge is that protecting all accesses to shared locations requires non-local reasoning: All control flow paths leading to a memory operation on shared data must be dominated by a synchronization operation. A data race may occur if the programmer forgets to synchronize even a single path. To make matters worse, even if every access to shared data is protected, the program may still end up in an inconsistent state due to a high-level data race [Artho et al. 2003]. This can occur when there exists a consistency relation between multiple memory locations and the programmer's use of synchronization fails to ensure that this relation is maintained at every instant. Analysis of real world software defects suggests that these kinds of races occur frequently [Lu et al. 2007; Lu et al. 2008]. Avoiding high-level data races requires the same kind of non-local reasoning but is further complicated by the fact that multiple locks may have to be acquired in a specific order.

Data-centric synchronization is a declarative approach to concurrency control first proposed by some of the present authors [Vaziri et al. 2006]. Data-centric synchronization advocates that instead of focusing on the flow of control, programmers should identify sets of memory locations that share some consistency property and group those locations in *atomic sets* that will be updated atomically. Programmers need not specify where or what kind of synchronization operations to insert; instead, each atomic set has an associated set of *units of work*, code fragments that preserve the consistency of their associated atomic set. Synchronization code is automatically generated by a compiler which is free to choose where and what type of synchronization to insert. Such a declarative approach has the benefit that it is possible to change the concurrency-control mechanism, e.g., going from standard locks to read/write locks or even to transactional memory, without changing the program's source code. In a data-centric approach, the non-local reasoning that permeates traditional approaches to synchronization is replaced by a focus on shared data. High-level data races are naturally avoided as an atomic set can protect multiple locations and multiple atomic sets can be manipulated atomically within the same unit of work.

The purpose of this paper is to evaluate the applicability and benefits of data-centric synchronization in the context of a mainstream object-oriented language. To this end, we have extended the Java programming language with language features for data-centric synchronization and implemented a compiler that synthesizes concurrency control operations. The changes to the source language are unintrusive, and are limited to five optional annotations on classes and variable declarations. Like Java, the resulting language permits separate compilation, and the compiled code is in the standard Java bytecode representation and is backwards compatible with plain Java. We refer to the extended language as AJ. The criteria which we consider in our evaluation are:

expressiveness. Are there significant limitations to the range of concurrent problems which can be solved with AJ?

programmer effort. How many program edits are required to make code thread-safe?

performance. How does the performance of code generated by our AJ compiler compare to that of traditional Java implementations?

While data-centric synchronization takes fine-grained control over placement and selection of synchronization operations from the programmer, and is thus possibly going to lead to reduced concurrency, it provides strong consistency guarantees. By making the tradeoff explicit, we allow programmers to make an informed choice between the two approaches.

In our previous work [Vaziri et al. 2006], we relied on static whole-program program analysis to infer where synchronization operations should be placed in order to ensure that units of work are serializable from the perspective of each atomic set, a property we call *atomic-set serializability*. Preliminary experiments suggested that atomic sets require fewer annotations than implementations based on synchronized blocks in Java while eliminating known concurrency-related errors [Wang and Stoller 2006b; Hammer et al. 2008]. However, while promising, the approach's reliance on whole-program analysis limited applicability and dimmed the prospects for adoption. Whole-program analysis is prohibitively expensive for large code bases and does not easily accommodate dynamic loading, native methods and reflection which are integral parts of the Java platform. Furthermore, that work did not support atomic sets spanning multiple objects which led to inefficient code.

In this paper we present a variant of the atomic sets model of Vaziri et al. [2006]. We introduce a new mechanism for constructing atomic sets that span multiple objects and for *internal* objects that provide strong encapsulation for data whose concurrency is managed externally. The new approach obviates the need for whole-program analysis with a type system that guarantees that any well-typed program is atomic-set serializable, which means that all operations performed on locations that belong to an atomic set are serializable. To empirically evaluate the applicability of our ideas on real-world code, we implemented AJ within the Eclipse development environment.

We then refactored classes from the Java Collections Framework and a set of Java applications that includes the SPECjbb performance benchmark into AJ, and measured annotation overhead. We found that the collection classes required approximately 40 annotations per KLOC, and that the annotation overhead for the other applications ranged from 0.6 to 11.5 annotations per KLOC. For each of the applications, we found that our data-centric approach required fewer annotations than the number of synchronized blocks that were present in the original Java code. A number of minor refactorings was needed to transform the subject programs into valid AJ programs, as will be discussed in Section 7. For example, in several of our subject programs, field accesses were replaced with calls to getter/setter methods, and calls to wait() and notify() were replaced with uses of condition variables on atomic sets, a feature that will be discussed in Section 6.

We also report on extensive performance measurements with AJ versions of the SPECjbb benchmark. While the version that we obtained by naively introducing atomic sets did not scale well, we were able to achieve nearly the same performance as the original Java version after some performance tuning, without affecting annotation overhead materially. Specifically, our *tuned* AJ version of SPECjbb achieves a throughput of 90.8% of that of the original Java implementation when run with 98 threads. We consider these results an indication that our approach is capable of generating code with acceptable performance while providing a correctness guarantee that Java's current synchronization mechanism does not offer. In summary, we make the following contributions:

— A data-centric approach to synchronization that permits separate compilation, multi-object atomic sets and strongly encapsulated objects.
— A formalization of the type system for a core calculus and a proof that any well-typed program is atomic-set serializable.
— A prototype implementation in a mainstream object-oriented language and an integration with a development environment.
— An empirical evaluation on several Java applications including widely used libraries and a well-known performance benchmark.

Our prototype implementation does not support multiple atomic sets and our type system does not deal with generics. Adding multiple atomic sets is simply a matter of engineering, we do not forsee any major challenges. Supporting generics would complicate the formal treatment without fundamentally affecting our results.

The remainder of this paper is organized as follows. Section 2 reviews related work on language designs and type systems that aim to prevent concurrency-related errors. Section 3 presents an informal overview of the AJ language, using several motivating examples. The implementation of AJ is presented in Section 5. Section 6 proposes a number of small extensions to the core AJ language, including a generalized form of the unitfor construct and condition variables. Section 7 presents an empirical evaluation of our language design, by measuring annotation overhead and performance. Finally, Section 8 presents conclusions and discusses possible avenues for future work.

2. BACKGROUND AND INFLUENCES

This paper builds on the atomic set programming model of Vaziri, Tip and Dolby [Vaziri et al. 2006]. That work also introduced a notion of problematic interleaving scenarios and then used this notion to define a correctness criterion, named atomic-set serializability, which rules out high-level data races. Subsequent work by a subset of the authors and by an unrelated group explored how to detect concurrency-related errors based on this criterion (statically [Kidd et al. 2011] and dynamically [Hammer et al. 2008; Lai et al. 2010]). Atomic sets share characteristics with data groups [Leino 1998] and regions [Greenhouse and Boyland 1999] which group mutable fields to enable modular verification and reasoning about program transformations. Like atomic sets, regions and groups may be extended in subclasses, but unlike atomic sets, both are hierarchical and regions overlap. Another data-centric approach was proposed by Ceze et al. [2008], with a sketch of a possible transactional memory implementation. Atomic sets can also be viewed as a generalization of Hoare monitors [Hoare 1974] to multiple objects. In particular, we provide two mechanisms, unitfor and aliasing, for merging distinct atomic sets, as well as a data-centric notion of condition variables. Bergan et al. [Bergan et al. 2010] proposed a hardware assisted data-centric atomicity violation detection and avoidance approach.

Data-centric concurrency control is but one alternative to explicit locking. Transactional memory [Herlihy and Moss 1993] approaches concurrency control from a database angle. Certain code fragments are specified to execute atomically, and it is up to the implementation to enforce mutual exclusion. While programmers need not worry about which data will be accessed in a transaction, they still have to identify where to place atomic sections and thus some of the same non-local reasoning as with synchronized statements is required. The main simplification is that it is not necessary to identify and name locks. Another way to avoid explicit locking is to perform lock inference. Like transactional memory, programmers must annotate programs with atomic sections, but instead of relying on a transactional memory mechanism, static analysis is used to determine which locks to acquire [Cherem et al. 2008; McCloskey et al. 2006]. While more efficient than transactions, as there is no need to support abort/undo semantics, lock inference relies on whole-program information and thus cannot deal with the dynamic features of Java.

Type systems for atomicity and race-freedom are another influence on our work. The type system of Abadi et al. [2006] guarantees the absence of data races. The general approach is to have a programmer provide redundant type annotations on top of a program with explicit lock operations. The type system thus only needs to check that the synchronization and the type annotations are consistent. In that approach, methods declare the locks they require and a guarded_by construct is used to indicate which lock protects a field. With 20 annotations per KLOC for the Java collections framework, the approach is relatively lightweight, but unlike atomic sets the programmer must add explicit synchronization to the code. Moreover, atomic-set serializability is a higher level property than data race freedom. The type system of Flanagan and Qadeer [Flanagan and Qadeer 2003; Flanagan et al. 2008] guarantees atomicity, i.e., equivalence to a serial execution. As above, fields are annotated with guarded_by or write_guarded_by to indicate that (write) access to the field must be protected by a lock. Methods are annotated with atomic to indicate their atomicity and with requires to indicate which locks must be held by callers. Atomic-set serializability recognizes some benign interleavings as correct that global serializability does not. Flanagan and Qadeer evaluated their type system on Java library classes and report an average of 23.3 annotations per KLOC of code. However, similar to the approach by Abadi et al. [Abadi et al. 2006] and unlike atomic sets, it is assumed that the programmer has added synchronization to the code. Inference [Flanagan et al. 2008] reduces the annotation burden. More recent work has looked at building atomicity [Kulkarni et al. 2010] and determinism [Bocchino et al. 2009] directly in the programming language.

Our type system was influenced by ownership type systems which started out as an attempt to control the sharing of references [Noble et al. 1998] and is typically used to enforce a strong form of encapsulation. Our treatment of internal objects is close to traditional ownership as all references to these objects are encapsulated. But unlike the early owner-as-dominator type systems [Clarke et al. 1998] there is no single access point. Indeed, in order to support iterators we have loosened the restriction of a single owner and allow the elements of atomic sets that are not part of internal classes to be viewed and manipulated from the outside. The ownership type system of Boyapati and Rinard [Boyapati and Rinard 2001] ensures that Java-like programs are data race-free. In that work, classes are parameterized with a list of owners and methods may require that their callers hold particular locks. A simple unification-based form of local type inference is used to reduce the annotation burden. While no direct comparison is possible as the implementation of Boyapati and Rinard [Boyapati and Rinard 2001] is not available, we believe atomic sets have lower annotation overhead overall, and are better integrated into Java. Deadlocks can also be ruled out by ownership type systems [Boyapati et al. 2002] but this comes at the price of expressiveness and an increased annotation burden. We feel that some form of static analysis may be a better fit to address deadlocks, but have left the matter to future work.

Attention to high-level data races is relatively recent. Many static [Engler and Ashcraft 2003; Leino et al. 1999] and dynamic race detectors [O'Callahan and Choi 2003; Savage et al. 1997], as well as type systems [Boyapati and Rinard 2001; Flanagan and Freund 2000b] that guarantee race freedom are based on the common definition of data races and therefore do not handle high-level races. An extension to ESC/Java detects a class of high-level data races, called "stale-value errors" [Burrows and Leino 2004]. The value of a local variable is stale if it is used beyond the critical section in which it was defined. View consistency [Artho et al. 2003] is a correctness criterion that ensures that multiple reads in a thread observe a consistent state. A view is defined to be the set of variables that a lock protects. Two threads are view consistent if all the views in the execution of one, intersected with the maximal view of the other, form a chain under set inclusion. View consistency can be checked dynamically [Artho et al. 2003] or statically [von Praun and Gross 2004]. In our approach, however, the programmer indicates explicitly what sets of locations form an atomic set, so this information does not need to be extracted from the locking structure of the code, which may not be correct. Recently, Lucia et al. [2010] presented an approach for detecting atomicity violations that involve multiple memory locations. In Lucia's work, related memory locations are identified by giving them the same color, and architectural support is proposed to implement the technique efficiently.

The Serializability Violation Detector [Xu et al. 2005] is a tool that dynamically infers atomic sections, based on data and control dependences, and then detects if these sections are non-serializable by checking a rule based on strict 2-Phase Locking. One of its key features is that it does not rely on the possibly buggy locking structure of the program to infer atomic sections. We share a similar viewpoint by having a definition of data races that does not rely on locks. The detector produces both false positives and false negatives, depending on the precision of the inferred atomic sections.

Deng et al. [2002] present a method that allows the user to specify synchronization patterns that are used to synthesize synchronized code. The generated code can then be verified using the Bandera toolset. In this approach, the user must specify explicitly the regions of code that need synchronization, but we do not require this. Unlike them, we focus on only one kind of synchronization pattern: exclusion between two regions that access the same atomic set.

3. DATA-CENTRIC SYNCHRONIZATION WITH AJ

AJ extends the syntax of the Java programming language with annotations needed to support the data-centric programming model of Vaziri et al. [2006]. An AJ class can have zero or more atomicset declarations. Each atomic set has a symbolic name and intuitively corresponds to a logical lock protecting a set of memory locations. Associated with each atomic set is a set of *units of work*, code fragments that, when executed sequentially, preserve the consistency of their associated atomic sets. By default, the units of work for an atomic set declared in a class C consist of all non-private methods in C and its subclasses. Given data-centric synchronization annotations, AJ infers the placement of concurrency control operations in such a way that units of work are serializable from the perspective of each atomic set, a property we call atomic-set serializability. The inferred synchronization ensures that any execution is equivalent to one in which, for each atomic set, its units of work occur in some serial order. One may think of a unit of work as being an atomic section [Harris and Fraser 2003] that is only atomic with respect to a particular set of memory locations. Accesses to locations not in the set are visible to other threads. The AJ implementation is free to choose the type of concurrency control operations and to optimize their placement. Thus, for instance, methods declared private or called through this usually do not require synchronization as their calling context has established atomicity. Methods that do not operate on locations that are within an atomic set will typically not be synchronized either.

Fig. 1 shows an integer counter class with atomic increment and decrement methods. Each instance of Counter has its own instance of its atomic set a. The locations protected by the atomic sets are identified by annotating the corresponding fields with atomic (a). Atomic set declarations are inherited by subclasses, so every instance of a subclass of Counter has its own a and can add some of its fields to the atomic set. AJ requires that fields belonging to an atomic set must be accessed through the (implicit) this reference. Note that this is a stronger property than labeling the field private, as in Java two instances of the same class can access each other's private fields.

```
class Counter {                          Counter c = new Counter();
    atomicset a;                         c.inc();
    atomic(a) int val;                   c.dec();
    int get() { return val; }            ...
    void dec() { val--; }
    void inc() { val++; }
}
```

Fig. 1: A simple counter class.

It is often the case that an atomic set must protect fields belonging to more than one object. While it is not possible to refer directly to another object's atomic set, AJ allows merging atomic sets using *aliasing* annotations. An atomic set a in an object pointed to by a variable x may be aliased with an atomic set b in the object pointed to by this by placing the alias annotation |a = this.b| on the declaration of x. This has the effect of merging the atomic sets in these objects. Fig. 2 shows a PairCounter class which has two integer counters, low and high, and a method, incHigh() that updates the difference between them. To this end, it introduces a new atomic set b for the diff field, and it aliases the atomic sets of the counters with b to form a single atomic set.

```
class PairCounter {
   atomicset b;
   atomic(b) int diff;
   Counter|a=this.b| low = new Counter|a=this.b|();
   Counter|a=this.b| high = new Counter|a=this.b|();
   void incHigh() { high.inc();   diff = high.get()-low.get(); }
   ...
}
```

Fig. 2: Aliased atomic sets.

There are cases where a method needs to coarsen the granularity of atomicity for some of its arguments. This is achieved by declaring additional units of work by annotating arguments with unitfor(a). If this annotation appears on some parameter p of some method m of a class D, this indicates that m is an additional unit of work for atomic set a of object p. Such cases—where a method is a unit of work for multiple atomic sets—are treated as if the method is a unit of work for the *union* of these atomic sets. Alias annotations have a similar effect. Fig. 3 illustrates this with a transfer method which must atomically update two Counter objects with different atomic sets.

```
class Transfer {
   void transfer(unitfor(a) Counter from, unitfor(a) Counter to) {   from.dec(); to.inc();   }
}
```

Fig. 3: Adding atomic sets to a unit of work using unitfor.

For performance reasons it may be advantageous to avoid synchronization around objects that are used to implement the representation of a given data structure. This is safe only if it is guaranteed that no reference to these representation objects ever leaks to clients where it could be manipulated without synchronization. The internal annotation is used to declare a class or interface and all of its subclasses as being private to a data structure. Internal classes must always have their atomic sets aliased to some enclosing data structure, which can be viewed as their "owner". The AJ type system enforces encapsulation of internal classes. The example of Fig. 4 illustrates the use of internal classes. Here, class Cell is internal. Class Main creates an instance of Cell, aliases its atomic set, b to its own atomic set a, and stores it in field c. Hence, the type system ensures that the Cell object will only be manipulated by the corresponding Main object.
It is noteworthy to observe that the internal annotation does not change the semantics of the application; its purpose is to enable the implementation to remove some redundant synchronization operations. While it would be possible to infer this annotation, doing so would require interprocedural analysis which we avoid in this work.

These AJ data-centric synchronization annotations are summarized in Fig. 5.

```
internal class Cell {
    atomicset b;    atomic(b) Object val;
    Object getset(Object o) { Object old = val; val = o; return old; }
}

class Main {
    atomicset a;    final Cell|b=this.a| c = new Cell|b=this.a|();
    void set(Object o) { c.getset(o); }
}
```

Fig. 4: An internal class.

3.1. Motivating Example

Fig. 6 shows some key fragments of a simplified version of the LinkedList class, a representative of the Java Standard Collections framework, made thread-safe using data-centric synchronization. The figure shows the abstract class AbsList which defines the interface of all lists and a concrete list, LinkedList. The designer of the abstract list has chosen to equip it with an atomic set a which is inherited by subclasses. Within AbsList the only field that needs protection is the integer size. It is annotated atomic(a) to denote that it belongs to a. The methods of AbsList and its subclasses are the units of work for a.

The method addAll(unitfor(a) AbsList c) must operate on multiple atomic sets, namely the receiver and the argument c. Logically, the list c must remain unchanged during the entire execution of addAll. By annotating parameter c with unitfor(a), we merge the

atomicset a	A class or interface declaration may have multiple atomic set declarations. Atomic sets are inherited and may be referenced in subclasses.
atomic(a)	Annotation on instance fields and classes. A field can belong to at most one atomic set. Annotated fields can only be accessed from the this reference. When added to a class declaration, this annotation is a shorthand for placing the same annotation on all instance fields in the class and its subclasses.
unitfor(a)	Each method argument can be annotated by one or more unitfor annotations. When the name is omitted, the annotated method becomes a unit of work for *all* atomic sets in the parameter object.
internal	Annotation on class declarations which must be preserved by inheritance. The type system tracks internal objects and ensures that no reference to an internal object can leak outside of the object that constructs it.
|a=this.b|	Annotation on variable declarations and in constructor expressions. This indicates that the atomic set a of the type of the annotated variable or constructed object is aliased with the current object's atomic set b.

Fig. 5: Data-centric annotations in AJ.

```
public abstract class AbsList {
 atomicset a;
 atomic(a) int size;
 public int size(){
    return size;
 }
 public abstract ListIterator iterator();
 public abstract void add(Object o);
 public abstract boolean
    addAll(unitfor(a) AbsList c);
 public abstract Object get(int i);
}

internal class Entry {
  atomicset b;
  atomic(b) Object elem;
  atomic(b) Entry next|b=this.b|;
  atomic(b) Entry prev|b=this.b|;
  ...
}
```

```
class LinkedList extends AbsList {
  atomic(a) Entry header|b=this.a|;
  public LinkedList() {
    header = new Entry|b=this.a|(null,null,null);
    header.next = header.prev = header;
  }
  public void add(Object o) {
    Entry newEntry|b=this.a| =
      new Entry|b=this.a|(o, header, header.prev);
    newEntry.prev.next = newEntry;
    newEntry.next.prev = newEntry;
    size++;
  }
  public ListIterator iterator() {
    return (ListIterator)
      new ListItr|l=this.a|(this,this.header, 0);
  }
  ... // other list methods
}
```

Fig. 6: AbsList, LinkedList and Entry classes

atomic set a in the receiver object with the atomic set a in the argument object for the duration of the method's execution.

In class LinkedList, the header field points to a doubly-linked list of Entry objects. LinkedList adds header to the atomic set a of its parent class to ensure that any method accessing both header and size will have a consistent view of these fields. However, note that the above is not sufficient for the data structure to be thread-safe: It is also necessary to protect the doubly-linked list itself. This requires defining an atomic set b in class Entry to protect the fields next and prev. Furthermore, units of work for the LinkedList object must encompass the units of work for the Entry objects it refers to. This is achieved by placing the alias annotation |b=this.a| on all allocation sites and variables of type Entry inside LinkedList to indicate that the atomic set b of these Entry objects should be combined with the list's atomic set a. Similar annotations, |b=this.b|, are placed on the fields next and prev of Entry. These imply that the atomic sets b of objects pointed to by these fields are merged with the atomic set b of this. Together with the annotation on header, they cause the entire backbone of the LinkedList to be in a single atomic set. Any unit of work for the list, including its Entry objects, will be performed atomically with respect to this merged atomic set. As an optimization, Entry is declared internal. This means that the type system will guarantee that no instance of Entry can be accessed without going through the methods of LinkedList. Thus, an implementation can omit synchronization for all of Entry's methods and leave concurrency control to the list object.

Each expression in our type system potentially has alias information. If there is no alias information, this means that either the expression represents an object that has no atomic sets, or that the object is an independent object that performs its own synchronization. The type system tracks aliasing annotations and prevents, e.g., the Entry object of one linked list from ending up within another linked list. Practically, this means that some types of casts are disallowed. Casting away an alias annotation (thus losing information) is allowed, but forging an alias annotation is not. For instance, the iterator() method creates an object of type ListItr (a class that is private to class LinkedList), which has an atomic set aliased to that of the linked list. This alias information is cast away in the return statement of the method.

A non-internal class such as LinkedList can be instantiated in two ways: new LinkedList() and new LinkedList|a=this.x|(). The former signifies a new instance of LinkedList that is responsible for its own synchronization, while the latter means that the atomic set of the new instance is the same as the atomic set x of the current object. The latter is especially useful when defining new data structures in terms of other data structures. For example, one could define a Stack in terms of a LinkedList and achieve correct synchronization behavior by having an atomic set in Stack that is aliased to the atomic set in the underlying LinkedList. This kind of compositionality is a key contribution of this paper and was not supported in the original work by Vaziri et al. [2006]. For internal classes such as Entry an aliased allocation site such as new Entry|b=this.a| is the only valid instantiation because an internal object must share the atomic set of its creator. As usual with type-based approaches, the bindings created by aliasing cannot be modified after creation.

3.2. Arrays

Arrays are fully handled by our implementation. Supporting arrays requires being able to specify atomicity constraints at three different levels. The declaration

```
atomic(a) B[] vals;
```

indicates that the reference to array vals is part of atomic set a, however the contents of the array can be updated without synchronization. The declaration

```
atomic(a) B[] vals|this.a[]|;
```

indicates that not only is the reference to the array to be accessed atomically, but the contents of the array are also part of atomic set a and must be accessed in a synchronized manner. Finally, the declaration

```
atomic(a) B[] vals|this.a[]b=this.a|;
```

indicates that, additionally, the atomic set b of each of the objects contained within the array should be merged with atomic set a. In our experience, we found all three of these forms of array annotation to be useful.

3.3. Data Races and Deadlocks

AJ does not completely prevent programmer errors. Data races can occur within a unit of work if the code manipulates data that is not part of the unit's atomic set. Thus it is incumbent on the programmer to correctly annotate all fields which share a consistency property, and to place unitfor annotations on method parameters as needed. Forgetting to annotate a field or method parameter can result in concurrency errors.

Our implementation of atomic set associates locks with atomic sets. There is thus the potential for deadlocks when multiple non-aliased atomic sets are manipulated by the same unit of work. We support a form of deadlock avoidance for methods that have unitfor annotations, by atomically acquiring the locks for all atomic sets that the method is a unit of work for. However, we cannot prevent deadlock when a thread executes a unit of work for some atomic set a that (transitively) invokes a unit of work for another atomic set b, and where another thread invokes a unit of work for atomic set b that (transitively) invokes a unit of work for atomic set a. In this respect, AJ programs are neither more nor less prone to deadlock than standard Java programs that acquire multiple locks out of order. We do, however, believe that the declarative nature of synchronization annotations in AJ simplifies the design of static analyses for

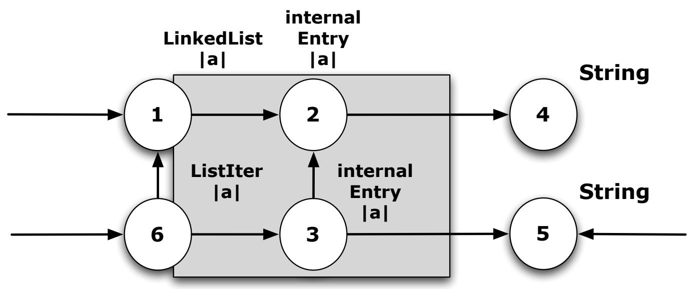

Fig. 7: Example. The instance 1 of LinkedList is the owner of the atomic set composed of objects 1, 2, 3 and 6. Since the two Entry objects are declared internal to the atomic set, the type system will ensure that no references to these object may be leaked outside of the atomic set. The ListIterator i (object 6) belongs to the atomic set but can also be accessed from the outside. The elements contained in the collection (4 and 5) are not protected by the atomic set and could potentially be modified concurrently.

detecting possible deadlocks, and this is a topic that we plan to investigate as future work.

3.4. Complete LinkedList example

Fig. 8 and Fig. 9 show the complete LinkedList example, including a small client. Fig. 7 illustrates the structure of the atomic sets in the example program. Notice that only a small number of data-centric synchronization annotations (highlighted) are needed to ensure correct synchronization behavior. Consider the call to the ListItr() constructor on line 34. The alias annotation |l=this.L| ensures that the atomic set l of ListItr is merged with this.L. The constructor is declared on line 59. It requires a LinkedList parameter l with an atomic set L that is merged with this.l. This alias annotation together with the one at the constructor call site, ensures that iterator() returns a ListItr object that corresponds to the list in question. Effectively, the methods in the iterator become additional units of work for L, and will provide the same atomicity constraints as any non-private method of the list. Notice that the return value of the iterator() method is cast to ListIterator (line 34). In our type system, there are no implicit casts, and therefore these upcasts must be applied explicitly. The ListItr constructor call results in an object with alias information |l=this.L|. This information must be erased explicitly with a cast before returning the object, since the return type has no alias information. It is a type error to erase the alias information of internal objects.

Finally, consider the Client class. The main() method first creates LinkedLists x, y, and z, and executes two threads that concurrently add the contents of the lists y ({a,a}) and z ({b,b}) to the list x. The client uses an iterator to traverse list x in the forward direction to replace each "b" with a "c". It then uses the same iterator to traverse the list in the backward direction to print the contents of each node in the list. This example was chosen to illustrate that our type system is capable of handling complex iterators that can modify the state of an underlying collection.

In the *absence of any synchronization* (i.e., if we assume that the highlighted code fragments have been omitted from the program), the execution of the two calls to addAll() on line 88 may be interleaved in arbitrary ways. As a result, the addition of the elements from the lists y and z to the list x may be intermixed, so that the list x may contain, for example, a, c, c, a, or c, a, a, c upon program termination. In fact, other interleavings exist in which the program terminates with a NullPointerException (e.g.,

```
class LinkedList extends AbsList {
   atomic(L) private Entry header|E=this.L| = new Entry|E=this.L|(null,null,null);
   public LinkedList() { header.next = header.prev = header; }
   public void add(Object o) {
      Entry newEntry|E=this.L| = new Entry|E=this.L|(o, header, header.prev);
      newEntry.prev.next = newEntry; newEntry.next.prev = newEntry; size++;
   }
   public Object get(int index) {
      if (index < 0 || index >= size()) throw new IndexOutOfBoundsException();
      Entry e|E=this.L| = header;
      for (int i = 0; i ¡= index; i++) e = e.next;
      return e.elem;
   }
   public boolean equals(unitfor Object o) {
      if (o == this) return true;
      if (!(o instanceof LinkedList)) return false;
      ListIterator e1 = iterator();
      ListIterator e2 = ((LinkedList) o).iterator();
      while (e1.hasNext() && e2.hasNext()) {
         Object o1 = e1.next(), o2 = e2.next();
         if (!(o1 == null ? o2 == null : o1.equals(o2))) return false;
      }
      return !(e1.hasNext() || e2.hasNext());
   }
   public int hashCode() {
      int hashCode = 1; ListIterator i = iterator();
      while (i.hasNext()) {
         Object obj = i.next();
         hashCode = 31 * hashCode + (obj == null ? 0 : obj.hashCode());
      }
      return hashCode;
   }
   public ListIterator iterator() {
      return (ListIterator) new ListItr|I=this.L|(this, this.header, 0);
   }
   public boolean addAll(unitfor(L) AbsList c) {
      boolean modified = false;
      ListIterator e = c.iterator();
      while (e.hasNext()) { add(e.next()); modified = true; }
      return modified;
   }
}
internal class Entry {
   atomicset E;
   atomic(E) Object elem;
   atomic(E) Entry next|E=this.E|;
   atomic(E) Entry prev|E=this.E|;
   Entry(Object elem, Entry next|E=this.E|, Entry prev|E=this.E|) {
      this.elem = elem; this.next = next; this.prev = prev;
   }
}
```

Fig. 8: Complete example program: LinkedList.

this may happen as a result of a thread being suspended in the middle of executing add(), when the `prev` and next pointers associated with the newly inserted list element are in an inconsistent state). We assume that it is the programmer's goal to ensure that all operations on lists are executed atomically. With the data-centric synchronization annotations, the two concurrent calls to addAll() happen atomically. Therefore, when

```
class ListItr implements ListIterator {
  atomicset l;
  atomic(l) private Entry lastReturned|E=this.l|;
  atomic(l) private Entry next|E=this.l|;
  atomic(l) private int nextIndex;
  atomic(l) final LinkedList list|L=this.l|;
  atomic(l) final Entry header|E=this.l|;

  ListItr(LinkedList l|L=this.l|, Entry h|E=this.l|, int index) {
    list = l; header = h; lastReturned = header;
    if (index < 0 || index > list.size()) throw new IndexOutOfBoundsException();
    next = header.next;
    for (nextIndex = 0; nextIndex < index; nextIndex++) next = next.next;
  }

  public boolean hasNext() { return nextIndex != list.size(); }

  public Object next() {
    if (nextIndex == list.size()) throw new NoSuchElementException();
    lastReturned = next; next = next.next; nextIndex++;
    return lastReturned.elem;
  }

  public boolean hasPrev() { return nextIndex != 0; }

  public Object prev() {
    if (nextIndex == 0) throw new NoSuchElementException();
    lastReturned = next = next.prev; nextIndex–;
    return lastReturned.elem;
  }
  public void set(Object o) {
    if (lastReturned == header) throw new IllegalStateException();
    lastReturned.elem = o;
  }
}

public class Client {
  public static void main(String[] args) throws Throwable {
    final AbsList x = new LinkedList();
    final AbsList y = new LinkedList();y.add(”a”);y.add(”a”);
    final AbsList z = new LinkedList();z.add(”b”);z.add(”b”);

    Thread t1 = new Thread(){ public void run(){ x.addAll(y); } };
    Thread t2 = new Thread(){ public void run(){ x.addAll(z); } };
    t1.start(); t2.start();
    t1.join(); t2.join();

    ListIterator it;
    for (it = x.iterator(); it.hasNext();){
      Object o = it.next(); if (o.equals(”b”)) it.set(”c”);
    }

    for ( ; it.hasPrev();) System.err.println(it.prev());
  } // can print aacc or ccaa, but not acac, caca, caac, acca
}
```

Fig. 9: Complete Example program: LinkedList.

the threads finish, the list x will contain either a, a, b, b, or b, b, a, a. Executing the remaining statements will result in replacing all b's with c's and printing the contents of the list in reverse order. Hence, the program will print c, c, a, a, or a, a, c, c. Since the program is properly synchronized, NullPointerExceptions cannot occur.

4. A FORMAL ACCOUNT OF AJ

We formalize AJ in a core calculus in the style of Wrigstad et al. [2009], which is an idealized version of Java extended with some of the key features of our proposal. The goal of the formalization is to prove soundness of the type system and illustrate its key properties. In particular, the type system ensures that references to instances of internal classes are encapsulated, and that atomic set aliasing constraints are preserved by reduction. This notion of correctness is expressed by the definition of well-formed configuration and run-time subtyping of Section 4.4. These properties allow us to show the soundness of an implementation that associates a single lock with all objects that have the same atomic set. The concurrency-control policy enforced by AJ is specified in Section 4.5 and a proof of atomic-set serializability is given in Section 4.6.

We focus on the essential features of AJ, namely atomic sets, atomic annotations on fields, alias annotations, and internal types. For simplicity, we restrict the formalization to a single atomic set per class, and exclude unitfor annotations. While both are important, they do not affect the type system which tracks aliases and internal classes. Adding multiple atomic sets would require a small change to the semantics which currently uses the addresses of objects as identifiers for atomic sets (instead, fresh values would have to be created for each atomic set). Adding unitfor would only require more complex traces; details are provided in Section 4.7. For brevity we omit orthogonal features of Java such as interfaces, control constructs, exceptions, final variables, primitive data types, arrays, generics, and thread creation and thread death. We start with a presentation of the syntax (Section 4.1), and static and dynamic semantics (Sections 4.2 and 4.3, resp.). Fig. 10 summarizes the main judgements of the static and dynamic semantics of the calculus, definitions are given in the corresponding subsections.

$\tau <: \tau'$	subtyping
cd OK	well-typed class
fd OK in C	well-typed method
md OK in C	well-typed method
$E \vdash \mathsf{s}$	well-typed statement
$H; \overline{T} \xrightarrow{\ell}_{\rho} H'; \overline{T'}$	reduction
$r <:^{r}_{H} \tau$	run-time subtyping
$H; T$ is WF	well-formed configuration
H' is WF in H	well-formed heap
T is WF in H	well-formed thread
F is WF in H	well-formed frame

Fig. 10: Summary of the main judgements used AJ's static and dynamic semantics.

4.1. Syntax

The AJ syntax is given in Fig. 11. In our core calculus fields are strongly private (they can only be accessed by dereferencing this) and methods are public. Without loss of generality, we use a “named form,” where the results of fields and variable accesses, method calls and instantiations must be immediately stored in a variable. A further simplification is the elimination of implicit upcasts for arguments, return values, and assignments. All casts are performed explicitly by cast statements which simplifies the other rules as they can assume type equality. Downcasts are safe in AJ because, as in Java, there is a run-time test to check that the object belongs to the target type. All

p	::=	$\overline{cd}$	*program*
cd	::=	ι class C extends D $\{as\ \overline{fd}\ \overline{md}\}$	*class*
as	::=	atomicset a $\mid \epsilon$	
fd	::=	$\alpha\ \tau$ f	*field*
md	::=	τ m ($\overline{\tau\ \mathsf{x}}$) {$\overline{\tau\ \mathsf{z}}$; s;return y}	*method*
s	::=	s;s $\mid$ skip $\mid$ x = this.f $\mid$ x = (τ)y $\mid$ this.f = z $\mid$ x = new τ () $\mid$ x = y.m ($\overline{\mathsf{z}}$)	*statement*
τ	::=	C|a = this.b| $\mid$ C	*type*
α	::=	atomic (a) $\mid \epsilon$	
ι	::=	internal $\mid \epsilon$	
E	::=	[] $\mid E[x : \tau]$	*type env*

Fig. 11: AJ's syntax. C, D are class names, f, m are field and method names, and x, y, z are names of variables or parameters. this is a distinguished variable. For simplicity, we assume that names of classes, fields, methods and variables are unique.

AJ-specific properties are preserved by subtyping, i.e., subtypes have the same atomic sets and are internal if their parent is internal. Upcasts are more interesting as they involve loss of type information. For brevity, we assume the existence of a well-formed class-table CT. Auxiliary functions are given in Fig. 12. We use the shorthand $\overline{x} <: \overline{\tau}$ to denote the pointwise subtype relation $x_1 <: \tau_1, \ldots, x_n <: \tau_n$. The subtyping relation is standard with the exception of the rule for types with alias annotations, which restricts subtyping to be annotation invariant.

$$\frac{\mathsf{C} <: \mathsf{D}}{\mathsf{C}|\mathsf{a}=\mathsf{this.b}| <: \mathsf{D}|\mathsf{a}=\mathsf{this.b}|}$$

We define the viewpoint adaption predicate *adapt* such that the value of $adapt(\tau, \tau')$ is the view of type τ from type τ'. If τ is a raw type C, then it is unchanged. If τ has an alias annotation, such as C|a = this.b|, and it is viewed from a type D|b = this.c|, then the value of this.b is substituted with this.c, yielding C|a = this.c|. In cases where *adapt* is undefined a type error will be reported as the type is not accessible from that particular viewpoint.

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

4.2. Type System

4.2.1. Classes, fields, and methods. A *class* definition C is well-typed if its fields are well-typed in the context of C. Furthermore, all methods (including non-overridden inherited methods) must be well-typed. In case the class inherits an atomic set, then it is not allowed to define a new one. If the class is declared internal it must have an atomic set, or inherit one. Finally, internal annotations must be preserved by inheritance. In the definitions below, we use the notation C *has* a to indicate that class C declares or inherits an atomic set a.

$$\frac{\begin{array}{c}\overline{fd}\ \text{OK in C} \quad methods(\mathsf{C}) = \overline{md'} \quad \overline{md'}\ \text{OK in C} \quad (\mathsf{D}\ has\ \mathsf{a}\quad implies \quad as = \epsilon) \\ (\iota = \mathsf{internal} \quad implies \quad \mathsf{C}\ has\ \mathsf{a}) \quad (\mathsf{D}\ is\ \mathsf{internal} \quad implies \quad \iota = \mathsf{internal})\end{array}}{\iota\ \mathsf{class\ C\ extends\ D}\ \{as\ \overline{fd}\ \overline{md}\}\ \text{OK}}\ \text{(T-CLASS)}$$

Subtyping:

$$\frac{}{\mathsf{C} <: \mathsf{C}} \qquad \frac{\mathsf{C}\ \mathsf{extends}\ \mathsf{D}}{\mathsf{C} <: \mathsf{D}} \qquad \frac{\mathsf{C} <: \mathsf{C}' \quad \mathsf{C}' <: \mathsf{D}}{\mathsf{C} <: \mathsf{D}}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{\mathsf{C}|\mathsf{a}=\mathsf{this.b}| <: \mathsf{D}|\mathsf{a}=\mathsf{this.b}|}$$

Extends:

$$\frac{CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{as\ \overline{fd}\ \overline{md}\}}{\mathsf{C}\ \mathsf{extends}\ \mathsf{D}}$$

Type lookup:

$$\frac{\tau\ \mathsf{m}(\overline{\tau_\mathsf{x}\,\mathsf{x}})\{\overline{\tau_\mathsf{z}\,\mathsf{z}};\ \mathsf{s}; \mathsf{return}\ \mathsf{y}\} \in methods(\mathsf{C})}{typeof(\mathsf{C.m}) = \overline{\tau_\mathsf{x}} \rightarrow \tau}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{as\ \overline{fd}\ \overline{md}\}\\ \mathsf{m}\ \textit{is not defined in}\ \overline{md}\end{array}}{typeof(\mathsf{C.m}) = typeof(\mathsf{D.m})}$$

$$\frac{\tau\ \mathsf{f} \in fields(\mathsf{C})}{typeof(\mathsf{C.f}) = \tau}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{as\ \overline{fd}\ \overline{md}\}\\ \mathsf{f}\ \textit{is not defined in}\ \overline{fd}\end{array}}{typeof(\mathsf{C.f}) = typeof(\mathsf{D.f})}$$

Local vars:

$$\frac{\begin{array}{c}H(F(\mathsf{this})) = \mathsf{C}|\omega|(\overline{r'})\\ mbody(\mathsf{C.m}) = (\overline{\tau_\mathsf{x}\,\mathsf{x}};\ \overline{\tau_\mathsf{z}\,\mathsf{z}}; \mathsf{s}; \mathsf{return}\ \mathsf{y})\\ E \equiv \overline{\mathsf{x} : \tau_\mathsf{x}}, \overline{\mathsf{z} : \tau_\mathsf{z}}, \mathsf{this} : \mathsf{C}\end{array}}{locals(\mathsf{m}, F) = E}$$

Method lookup:

$$\frac{\tau\ \mathsf{m}(\overline{\tau_\mathsf{x}\,\mathsf{x}})\{\overline{\tau_\mathsf{z}\,\mathsf{z}};\ \mathsf{s}; \mathsf{return}\ \mathsf{y}\} \in methods(\mathsf{C})}{mbody(\mathsf{C.m}) = (\overline{\tau_\mathsf{x}\,\mathsf{x}};\ \overline{\tau_\mathsf{z}\,\mathsf{z}};\ \mathsf{s}; \mathsf{return}\ \mathsf{y})}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\{as\ \overline{fd}\ \overline{md}\}\\ \mathsf{m}\ \textit{not in}\ \overline{md}\end{array}}{mbody(\mathsf{C.m}) = mbody(\mathsf{D.m})}$$

Internal lookup:

$$\frac{CT(\mathsf{C}) = \mathsf{internal}\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{\ldots\}}{\mathsf{C}\ is\ \mathsf{internal}}$$

Fields lookup:

$$\frac{}{fields(\mathsf{Object}) = \epsilon}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\{as\ \overline{fd}\ \overline{md}\}\\ fields(\mathsf{D}) = \overline{fd'}\end{array}}{fields(\mathsf{C}) = \overline{fd'}\ \overline{fd}}$$

Methods lookup:

$$\frac{}{methods(\mathsf{Object}) = \epsilon}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\{as\ \overline{fd}\ \overline{md}\}\\ methods(\mathsf{D}) = \overline{md'} \quad \overline{md''} = \overline{md'} - \overline{md}\end{array}}{methods(\mathsf{C}) = \overline{md}\ \overline{md''}}$$

Valid Method overriding:

$$\frac{\begin{array}{c}typeof(\mathsf{C.m}) = \overline{\tau'} \rightarrow \tau' \quad implies\\ \overline{\tau} = \overline{\tau'}\ and\ \tau = \tau'\end{array}}{override(\mathsf{m}, \mathsf{C}, \overline{\tau} \rightarrow \tau)}$$

Atomic set lookup:

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{as\ \overline{fd}\ \overline{md}\}\\ as = \epsilon \qquad \mathsf{D}\ has\ \mathsf{a}\end{array}}{\mathsf{C}\ has\ \mathsf{a}}$$

$$\frac{\begin{array}{c}CT(\mathsf{C}) = \iota\ \mathsf{class}\ \mathsf{C}\ \mathsf{extends}\ \mathsf{D}\ \{as\ \overline{fd}\ \overline{md}\}\\ as = \mathsf{atomicset}\ \mathsf{a}\end{array}}{\mathsf{C}\ has\ \mathsf{a}}$$

Atomic lookup:

$$\frac{\mathsf{atomic}(\mathsf{a})\ \tau\ \mathsf{f} \in fields(\mathsf{C})}{\mathsf{C.f}\ is\ \mathsf{atomic}}$$

Fig. 12: Auxiliary definitions.

Atomic sets referred to in *field* declarations must exist.

$$\frac{(\tau \equiv \mathsf{D}|\mathsf{a}{=}\mathsf{this.b}| \;\; \textit{implies} \;\; (\mathsf{D}\ \textit{has}\ \mathsf{a})\ \textit{and}\ (\mathsf{C}\ \textit{has}\ \mathsf{b})) \quad (\alpha = \mathsf{atomic}\,(\mathsf{a}) \;\; \textit{implies} \;\; \mathsf{C}\ \textit{has}\ \mathsf{a})}{\alpha\,\tau\,\mathsf{f} \quad \text{OK in } \mathsf{C}}\ \text{(T-FIELD)}$$

Checking a *method* requires typing its body in an environment E constructed by composing the disjoint sets of parameters, $\overline{\mathsf{x}}$, local variables, $\overline{\mathsf{z}}$ and the distinguished variable this. If class C has an atomic set, the type of this is $\mathsf{C}|\mathsf{a} = \mathsf{this.a}|$; This is the default case when an object is in charge of its own synchronization (i.e., its atomic set has not been aliased) and is needed to ensure that *adapt* is defined. The type of the local variable y appearing in the return statement must match the return type of the method, and if the method overrides an inherited method, the signature must be unchanged.

$$\frac{\begin{array}{c} E \equiv \overline{\mathsf{x} : \tau_{\mathsf{x}}}, \overline{\mathsf{z} : \tau_{\mathsf{z}}}, \mathsf{this} : \tau_{\mathsf{this}} \quad E \vdash \mathsf{s}; \mathsf{return}\,\mathsf{y} \quad E(\mathsf{y}) = \tau \quad \mathsf{C}\ \mathsf{extends}\ \mathsf{D} \\ (\textit{if}\ \mathsf{C}\ \textit{has}\ \mathsf{a}\ \ \textit{then}\ (\tau_{\mathsf{this}} \equiv \mathsf{C}|\mathsf{a}{=}\mathsf{this.a}|)\ \textit{else}\ (\tau_{\mathsf{this}} \equiv \mathsf{C})) \quad \textit{override}(\mathsf{m}, \mathsf{D}, \overline{\tau_{\mathsf{x}}} \rightarrow \tau) \end{array}}{\tau\ \mathsf{m}(\overline{\tau_{\mathsf{x}}\,\mathsf{x}})\{\overline{\tau_{\mathsf{z}}\,\mathsf{z}};\ \mathsf{s}; \mathsf{return}\,\mathsf{y}\} \quad \text{OK in } \mathsf{C}}\ \text{(T-METHOD)}$$

Observant readers will note that we are checking inherited methods with the type of this bound to the subclass C and not to the defining class of the method (we are using the dynamic type of this). This prevents the implicit upcast in method invocation from being used to subvert the type system. Consider the following program which, without the above treatment of inherited methods, would leak a reference to an internal object.

```
class Id extends Object {
  Id id() {
    Id x;
    x = this;
    return x;
  }
}

internal class E extends Id {
  atomicset a;
}
```

```
class C extends Object {
  atomicset b;
  Id m() {
    E|a=this.b| y;
    Id z;
    y = new E|a=this.b|();
    z = y.id();
    return z;
  }
}
```

The instance of E is an internal class and should remain private to its owner (an instance of class C). Yet, if the invocation of id() were allowed, it would be possible to pass off the E object as an Id which is not protected. In our type system the assignment x=this does not type check in the context of class E. This problem is standard in ownership type systems. One could avoid type-checking inherited methods repeatedly by declaring inherited methods *anonymous*, i.e., that they do not leak the this reference [Vitek and Bokowski 2001] or inferring the property by whole program analysis as in the work by Grothoff et al. [2007]. In AJ, the only methods that need this are methods inherited by an internal class.

4.2.2. Statements. There are two type rules for object creation. The first rule, (T-NEW-RAW), covers the case where the object being created is not annotated with an alias. If class C has an atomic set, this means we are requesting the construction of an object that can take care of its own synchronization. The only restriction that must be enforced in this case is that the class not be declared internal as internal classes always depend on an owner. The second rule, (T-NEW-ASET), covers the case when a C object is created with an alias $|\mathsf{a} = \mathsf{this.b}|$. In this case, we check that C indeed has an atomic set a and that this refers to an object which has an atomic set b.

$$\frac{\begin{array}{c}E(\mathsf{x}) = \mathsf{C} \\ \mathsf{C}\ \mathit{not}\ \mathsf{internal}\end{array}}{E \vdash \mathsf{x} = \mathsf{new\ C()}}\ \text{(T-NEW-RAW)} \qquad \frac{\begin{array}{c}E(\mathsf{x}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \\ \mathsf{C}\ \mathit{has}\ \mathsf{a} \quad E(\mathsf{this})\ \mathit{has}\ \mathsf{b}\end{array}}{E \vdash \mathsf{x} = \mathsf{new\ C}|\mathsf{a}{=}\mathsf{this.b}|()}\ \text{(T-NEW-ASET)}$$

There are three type rules for upcasts. (T-CAST-PLAIN) covers the case where neither type has an alias annotation. Rule (T-CAST-ASET) allows annotation invariant upcasts. Finally, (T-CAST-OFF) strips the annotation from a type. This is only allowed for non-internal classes.

$$\frac{E(\mathsf{x}) = \mathsf{D} \qquad E(\mathsf{y}) = \mathsf{C} \qquad \mathsf{D} <: \mathsf{C}}{E \vdash \mathsf{y} = (\mathsf{C})\mathsf{x}}\ \text{(T-CAST-PLAIN)} \qquad \frac{\begin{array}{c}E(\mathsf{x}) = \mathsf{D}|\mathsf{a}{=}\mathsf{this.b}| \qquad E(\mathsf{y}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \\ \mathsf{C}\ \mathit{has}\ \mathsf{a} \qquad E(\mathsf{this})\ \mathit{has}\ \mathsf{b} \qquad \mathsf{D} <: \mathsf{C}\end{array}}{E \vdash \mathsf{y} = (\mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|)\mathsf{x}}\ \text{(T-CAST-ASET)}$$

$$\frac{E(\mathsf{x}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \qquad \mathsf{C}\ \mathit{not}\ \mathsf{internal} \qquad E(\mathsf{y}) = \mathsf{C}}{E \vdash \mathsf{y} = (\mathsf{C})\mathsf{x}}\ \text{(T-CAST-OFF)}$$

The rule for method calls, (T-CALL), checks the types of the arguments and the return type. Viewpoint adaption is necessary to ensure that the types of the arguments and the return value are visible from the viewpoint of the receiver.

$$\frac{\begin{array}{c}E(\mathsf{y}) = \tau_{\mathsf{y}} \qquad \mathit{typeof}(\tau_{\mathsf{y}}.\mathsf{m}) = \overline{\tau} \rightarrow \tau \qquad E(\overline{\mathsf{z}}) = \overline{\tau_{\mathsf{z}}} \\ \overline{\tau_{\mathsf{z}}} = \mathit{adapt}(\overline{\tau}, \tau_{\mathsf{y}}) \qquad \tau' = \mathit{adapt}(\tau, \tau_{\mathsf{y}}) \qquad E(\mathsf{x}) = \tau'\end{array}}{E \vdash \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}})}\ \text{(T-CALL)}$$

Consider for instance calls (1) and (2) to method m() in the example below. The return type of m is $\tau \equiv \mathsf{C}|\mathsf{c}{=}\mathsf{this.a}|$. At (1) $\tau_{\mathsf{y}} \equiv \mathsf{A}|\mathsf{a}{=}\mathsf{this.b}|$, the value of $\mathit{adapt}(\tau, \tau_{\mathsf{y}}) = \mathsf{C}|\mathsf{c} = \mathsf{this.b}|$ indicating, as expected, that the C object shares the same atomic set as the receiver. On the other hand, a2 is created with its own atomic set. Thus, at (2), the result of $\mathit{adapt}(\tau, \mathsf{A})$ is undefined. The call does not type check because it would return a value with an unknown alias.

```
class A extends Object {                class B extends Object {
   atomicset a;                            atomicset b;
   C|c=this.a| m(){                        A f() {
      C|c=this.a| x;                          A|a=this.b| a1; C|c=this.b| c1; A a2;
      x=new C|c=this.a|();                    a1 = new A|a=this.b|();
      return x;                               c1 = a1.m();             //(1) OK
   }                                          a2 = new A();
}                                             c1 = a2.m();             //(2) ERROR
class C extends Object {                      return a2;
   atomicset c;                            }
}                                       }
```

The rules for field selection and update check that the type of the field matches that of the variable it is stored into.

$$\textsc{(t-select)}\quad \frac{E(\mathsf{this}) = \tau \quad E(\mathsf{x}) = \tau_{\mathsf{f}} \qquad \mathit{typeof}(\tau.\mathsf{f}) = \tau_{\mathsf{f}}}{E \;\vdash\; \mathsf{x} = \mathsf{this.f}} \qquad \textsc{(t-update)}\quad \frac{E(\mathsf{this}) = \tau \quad E(\mathsf{y}) = \tau_{\mathsf{f}} \qquad \mathit{typeof}(\tau.\mathsf{f}) = \tau_{\mathsf{f}}}{E \;\vdash\; \mathsf{this.f} = \mathsf{y}}$$

4.3. Dynamic Semantics

We formulate AJ's dynamic semantics as a small-step operational semantics. Fig. 13 shows the syntax used for heaps, threads, stacks, frames, and objects. An AJ configuration $H; \overline{T}$ consists of a single heap H of locations mapped to objects and a collection of threads $\overline{T}$. Each thread T has its own stack S, plus a unique thread id denoted ρ. A stack S is a sequence of triples $\langle \mathsf{m}\, F\;\; \mathsf{s}\rangle$ consisting of a method name m, a stack frame F mapping variables to locations, and a statement s. At run-time, an object $\mathsf{C}|\omega|(\overline{r})$, consists of a class C, an atomic set owner ω (either a location r or empty) and values $\overline{r}$ for the object's fields (either locations or null).

H	::=	$[] \mid H[r \mapsto v]$	*heap*	F	::=	$[] \mid F[\mathsf{y} \mapsto r]$	*stack frame*
T	::=	$\rho\, S \mid \rho\,\mathsf{NPE}$	*thread*	v	::=	$\mathsf{C}|\omega|(\overline{r})$	*object*
S	::=	$\epsilon \mid S\,\langle \mathsf{m}\, F\;\; \mathsf{s}\rangle$	*stack*	ω	::=	$r \mid \epsilon$	*owner atomic set*

Fig. 13: Syntax for heaps, threads, stacks, frames and objects.

We model multi-threaded Java programs with a fixed set of threads, $\overline{T}$, each of which initially starts with a call to a run method. Threads are terminated either when the run method returns or by a null pointer exception (NPE). The reduction relation $\xrightarrow{\ell}_{\rho}$ represents a step of evaluation. The label ℓ describes the action and the thread identifier ρ specifies the thread that performed it. Action labels can be one of the following: $\uparrow r.\mathsf{f}$ (field select), $\downarrow r.\mathsf{f}$ (field update), $\leftarrow r.\mathsf{m}$ (method return), $\rightarrow r.\mathsf{m}$ (method call), or ϵ (empty action). Labels will be used in Section 4.5 to define traces; they record operations that may lead to a data race (reads/writes) and operations that correspond to potential unit of work boundaries (calls/returns). Basic thread-scheduling is modeled as a non-deterministic choice in (D-SCHEDULE). Given a set of threads $\overline{T}\, T\, \overline{T'}$, the rule picks randomly one of the threads, T, for reduction.

$$\frac{\overset{\text{(D-SCHEDULE)}}{H;\overline{T}\,\overline{T'}\,T \xrightarrow{\ell}_{\rho} H';\overline{T}\,\overline{T'}\,T'}}{H;\overline{T}\,T\,\overline{T'} \xrightarrow{\ell}_{\rho} H';\overline{T}\,\overline{T'}\,T'}$$

We abuse syntax a little bit and treat return y as a statement. Returning from a call implies popping the topmost frame off the stack, and capturing the return value. Upcasts and skip statements have the expected semantics.

$$\frac{\overset{\text{(D-RETURN)}}{F(\mathsf{y}) = r \qquad F(\mathsf{this}) = r'}}{H;\overline{T}\,\rho\,S\,\langle \mathsf{m}'\,F'\ \ \mathsf{x} = \mathsf{y}'.\mathsf{m}(\overline{\mathsf{z}});\mathsf{s}'\rangle\langle \mathsf{m}\,F\ \ \mathsf{return}\,\mathsf{y}\rangle \xrightarrow{\leftarrow r'.\mathsf{m}}_{\rho} H;\overline{T}\,\rho\,S\,\langle \mathsf{m}'\,F'[\mathsf{x}\mapsto r]\ \ \mathsf{s}'\rangle}$$

$$\frac{\text{(D-CAST)}}{H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{x}{=}(\tau)\mathsf{y};\mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F[\mathsf{x}\mapsto F(\mathsf{y})]\ \ \mathsf{s}\rangle}$$

Field selection extracts one of the references stored in the object, while field update modifies the content of the object at the proper location. We define $H(r.\mathsf{f}_i)$ as follows: $H(r.\mathsf{f}_i) = r_i$ if $H(r) = \mathsf{C}|\omega|(r_1 \ldots r_i \ldots, r_n)$ and $\mathit{fields}(\mathsf{C}) = \mathsf{f}_1, \ldots \mathsf{f}_i \ldots, \mathsf{f}_n$.

$$\frac{\overset{\text{(D-SELECT)}}{F(\mathsf{this}) = r \qquad H(r.\mathsf{f}_i) = r_i}}{H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{x}{=}\mathsf{this}.\mathsf{f}_i;\mathsf{s}\rangle \xrightarrow{\uparrow r.\mathsf{f}_i}_{\rho} \quad H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F[\mathsf{x}\mapsto r_i]\ \ \mathsf{s}\rangle}$$

$$\frac{\overset{\text{(D-UPDATE)}}{F(\mathsf{this}) = r \qquad F(\mathsf{x}) = r_{\mathsf{x}} \qquad H(r) = \mathsf{C}|\omega|(\overline{r}, r_i, \overline{r'}) \qquad H' \equiv H[r \mapsto \mathsf{C}|\omega|(\overline{r}, r_{\mathsf{x}}, \overline{r'})]}}{H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{this}.\mathsf{f}_i{=}\mathsf{x};\mathsf{s}\rangle \xrightarrow{\downarrow r.\mathsf{f}_i}_{\rho} H';\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{s}\rangle}$$

Object creation comes in three flavors. (D-NEW-PLAIN) covers the construction of plain Java objects where the owner is empty. (D-NEW-SELF) takes care of creation of an instance of a class that has an atomic set and for which no alias annotation is specified. In this case, the owner is the newly created object itself. Lastly, (D-NEW-ALIAS) is for the construction of objects which have an alias annotation of the form $|\mathsf{a} = \mathsf{this}.\mathsf{b}|$. For those, we look up the owner of this and set it as the owner of the newly created object.

$$\frac{\overset{\text{(D-NEW-PLAIN)}}{\begin{array}{c} v \equiv \mathsf{C}|\epsilon|(\mathsf{null}_1 ... \mathsf{null}_n) \qquad r \text{ is fresh} \qquad \mathit{not}\ \mathsf{C}\ \mathit{has}\ \mathsf{a} \\ H' \equiv H[r \mapsto v] \qquad \mathit{length}(\mathit{fields}(\mathsf{C})) = n \qquad F' \equiv F[\mathsf{x} \mapsto r] \end{array}}}{H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{x} = \mathsf{new}\,\mathsf{C}();\mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H';\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F'\ \ \mathsf{s}\rangle}$$

$$\frac{\overset{\text{(D-NEW-SELF)}}{\begin{array}{c} v \equiv \mathsf{C}|r|(\mathsf{null}_1 ... \mathsf{null}_n) \qquad r \text{ is fresh} \qquad \mathsf{C}\ \mathit{has}\ \mathsf{a} \\ H' \equiv H[r \mapsto v] \qquad \mathit{length}(\mathit{fields}(\mathsf{C})) = n \qquad F' \equiv F[\mathsf{x} {\mapsto} r] \end{array}}}{H;\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{x}{=}\mathsf{new}\,\mathsf{C}();\mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H';\overline{T}\,\rho\,S\,\langle \mathsf{m}\,F'\ \ \mathsf{s}\rangle}$$

$$\frac{\begin{array}{c}\text{(D-NEW-ALIAS)}\\ v \equiv \mathsf{C}|r'|(\mathsf{null}_1 ... \mathsf{null}_n) \qquad r \text{ is fresh} \qquad \mathsf{C}\ \mathit{has}\ \mathsf{a} \quad \mathsf{D}\ \mathit{has}\ \mathsf{b}\\ H' \equiv H[r \mapsto v] \qquad |\mathit{fields}(\mathsf{C})| = n \qquad H(F(\mathsf{this})) = \mathsf{D}|r'|(\overline{r})\end{array}}{H; \overline{T}\, \rho\, S\, \langle \mathsf{m}\ F\ \ \mathsf{x} = \mathsf{new}\ \mathsf{C}|\mathsf{a} = \mathsf{this.b}|(); \mathsf{s}\rangle \xrightarrow{\epsilon}_\rho H'; \overline{T}\, \rho\, S\, \langle \mathsf{m}\ F[\mathsf{x} \mapsto r]\ \ \mathsf{s}\rangle}$$

Method calls push a new frame on the stack with local variables initialized to null and parameters bound to corresponding arguments. For brevity, null-pointer exceptions cause threads to immediately get stuck. More accurate treatment of exceptions (e.g., catch-blocks and stack unwinding) is unnecessary for the problem at hand.

$$\frac{\begin{array}{c}\text{(D-CALL)}\\ F(\mathsf{y}) = r \quad F(\overline{\mathsf{z}}) = \overline{r} \qquad H(r) = \mathsf{C}|\omega|(\overline{r'}) \qquad \mathit{mbody}(\mathsf{C.m}) = (\overline{\tau_\mathsf{x}\, \mathsf{x}'};\ \overline{\tau_\mathsf{y}\, \mathsf{y}}; \mathsf{s}'; \mathsf{return}\ \mathsf{y}')\\ F' \equiv [\overline{\mathsf{y} \mapsto \mathsf{null}}][\overline{\mathsf{x}' \mapsto r}][\mathsf{this} \mapsto r] \qquad S' \equiv S\, \langle \mathsf{m}'\ F\ \ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \langle \mathsf{m}\ F'\ \ \mathsf{s}'; \mathsf{return}\ \mathsf{y}'\rangle\end{array}}{H; \overline{T}\, \rho\, S\, \langle \mathsf{m}'\ F\ \ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \xrightarrow{\rightarrow r.\mathsf{m}}_\rho H; \overline{T}\, \rho\, S'}$$

$$\frac{\text{(D-CALL-NPE)}}{H; \overline{T}\, \rho\, S\, \langle \mathsf{m}'\ F[\mathsf{y} \mapsto \mathsf{null}]\ \ \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \xrightarrow{\epsilon}_\rho H; \overline{T}\, \rho\, \mathsf{NPE}}$$

4.4. Properties

We now proceed to establish preservation and progress for our type system. As usual the proofs rely on a notion of well-formed heaps, threads and configurations as well as run-time subtyping. We start with these auxiliary definitions. In a heap H, let $\mathit{owner}_H(r) = \omega$, if $H(r) = \mathsf{C}|\omega|(\overline{r})$. Let $\mathit{internal}_H(r)$ hold if $H(r) = \mathsf{C}|\omega|(\overline{r})$ and C *is* internal. We write τ *is raw* to mean that type τ is of the form C and has no alias annotation and τ *not raw* is the negation of τ *is raw*.

4.4.1. Run-time Subtyping Relation. The run-time subtyping relation, $r <:_H^{r_o} \tau$ indicates that a reference r is an instance of type τ at run-time, in the context of a reference r_o and a heap H. Since types may contain alias annotations that refer to this, we need a reference r_o to give meaning to this. There are three cases: (i) if $H(r)$ is null then the relation holds for all τ, (ii) if $H(r)$ is $\mathsf{C}|\omega|(\overline{r})$ then if τ is a raw type, D, the relation holds if C <: D and if C is not an internal class (to prevent leaking an internal object), and (iii) if τ is an aliased type D|a = this.b|, we must check that r has the same owner as r_o.

$$\frac{}{\mathsf{null} <:_H^{r_o} \tau} \qquad \frac{\begin{array}{c}H(r) = \mathsf{C}|\omega|(\overline{r}) \qquad \mathsf{C} <: \mathsf{D}\\ \mathsf{C}\ \mathit{not}\ \mathsf{internal}\end{array}}{r <:_H^{r_o} \mathsf{D}} \qquad \frac{\begin{array}{c}H(r) = \mathsf{C}|\omega|(\overline{r}) \qquad \mathsf{C} <: \mathsf{D}\\ \mathit{owner}_H(r) = \mathit{owner}_H(r_o)\end{array}}{r <:_H^{r_o} \mathsf{D}|\mathsf{a} = \mathsf{this.b}|}$$

Notice that the run-time subtyping relation satisfies the following property. If $r <:_H^{r_o} \tau$ and $r \neq \mathsf{null}$, then if τ ***is raw*** then $\mathit{not}\ \mathit{internal}_H(r)$, and if τ ***not raw*** then $\mathit{owner}_H(r) = \mathit{owner}_H(r_o)$.

4.4.2. Well-formed configurations. A ***configuration*** is well-formed, written $H; \overline{T}$ is WF, if the heap and threads are well-formed and the class table is well-typed, written $\vdash CT$. A heap H is well-formed if it is empty or if all fields of all objects it contains are well-typed, meaning that the reference corresponding to each field is a run-time subtype of the static type of that field. A thread T is well-formed, written T is WF in H, if it is stuck on a null pointer exception. Otherwise, a thread is well-formed if the topmost frame is well-formed, and if the remainder of the stack is well-formed. If the receiver

$$\frac{H \text{ is WF in } H \quad \overline{T} \text{ is WF in } H \quad \vdash CT}{H; \overline{T} \text{ is WF}}\ \text{(WF-CONFIGURATION)}$$

$$\frac{}{[] \ \text{ is WF in } H}\ \text{(WF-EMPTY-HEAP)} \qquad \frac{}{\rho\, \mathsf{NPE} \text{ is WF in } H}\ \text{(WF-NPE-THREAD)}$$

$$\frac{\langle \mathsf{run}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H \qquad \mathit{not\ internal}_H(F(\mathsf{this}))}{\rho\, \langle \mathsf{run}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H}\ \text{(WF-THREAD-BOT)}$$

$$\frac{\begin{array}{c}\langle \mathsf{m}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H \qquad \rho\, S \text{ is WF in } H \\ S \equiv S'\langle \mathsf{m}'\, F' \ \ s'\rangle \qquad s' \equiv \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}'}); \mathsf{s}'' \qquad \mathit{not\ internal}_H(F(\mathsf{this}))\end{array}}{\rho\, S\langle \mathsf{m}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H}\ \text{(WF-THREAD-NOT-INT)}$$

$$\frac{\begin{array}{c}\langle \mathsf{m}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H \quad \rho\, S \text{ is WF in } H \\ S \equiv S'\langle \mathsf{m}''\, F'' \ \ s''\rangle\langle \mathsf{m}_0\, F_0 \ \ s_0\rangle \ldots \langle \mathsf{m}_n\, F_n \ \ s_n\rangle \qquad s_n \equiv \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}'}); \mathsf{s}'' \\ \mathit{owner}_H(F''(\mathsf{this})) = \mathit{owner}_H(F(\mathsf{this})) = \ldots = \mathit{owner}_H(F_n(\mathsf{this})) \\ \mathit{internal}_H(F(\mathit{this})) \qquad \mathit{not\ internal}_H(F''(\mathsf{this})) \\ \mathit{internal}_H(F_0(\mathsf{this})) \ldots \mathit{internal}_H(F_n(\mathsf{this}))\end{array}}{\rho\, S\langle \mathsf{m}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H}\ \text{(WF-THREAD-INT)}$$

$$\frac{(\mathsf{C}\ \mathit{has}\ \mathsf{a}\ \ \mathit{implies}\ \ \omega \neq \epsilon) \qquad H' \text{ is WF in } H \qquad \mathit{fields}(\mathsf{C}) = \overline{\alpha\, \tau\, \mathsf{f}} \qquad \overline{r_{\mathsf{z}} <:^{r}_{H} \tau}}{H'[r \mapsto \mathsf{C}|\omega|(\overline{r_{\mathsf{z}}})] \ \text{ is WF in } H}\ \text{(WF-HEAP)}$$

$$\frac{\mathit{locals}(\mathsf{m}, F) = E \qquad E \vdash \mathsf{s} \qquad \forall\, \mathsf{x} \in \mathit{dom}(F), F(\mathsf{x}) <:^{F(\mathsf{this})}_{H} E(\mathsf{x})}{\langle \mathsf{m}\, F \ \ \mathsf{s}\rangle \text{ is WF in } H}\ \text{(WF-FRAME)}$$

Fig. 14: Well-formedness rules.

of the topmost stack frame is an instance of a class annotated as internal, then the remainder of the stack may have zero or more frames with internal receivers followed by at least one frame with a non-internal receiver, and the owners of the receivers of all the frames must be identical. A frame F is well-formed if for each variable x in the domain of F, the corresponding reference is a run-time subtype of the static type of x. The rules appear in Fig. 14.

4.4.3. Type Soundness. We prove type soundness of AJ by showing preservation and progress. Here, preservation means that reduction of a well-formed configuration results in a well-formed configuration, and the proof of preservation states that after a step of reduction a well-formed configuration remains well-formed.

We first define the notion of an *active* thread as a thread that has not stumbled on an NPE or returned from its bottommost stack frame.

Definition 4.1. A thread $T \equiv \rho\, S$ is *active*, denoted $active(T)$, if $S \not\equiv \mathsf{NPE}$ and $S \not\equiv \langle \mathsf{run}\, F\ \ \mathsf{return\ y}\rangle$.

For simplicity, the proof will assume that the statements of Fig. 11 include the expression $\mathsf{return\ y}$.

THEOREM 4.2. *Preservation. If* $H; \overline{T}\, T\, \overline{T'}$ *is WF and* $H; \overline{T}\, T\, \overline{T'} \xrightarrow{\ell}_{\rho} H'; \overline{T}\, \overline{T'}\, T'$, *then* $H; \overline{T}\, \overline{T'}\, T'$ *is WF.*

PROOF. We proceed by structural induction on the derivation of $H; \overline{T}\, T\, \overline{T'} \xrightarrow{\ell}_{\rho} H'; \overline{T}\, \overline{T'}\, T'$ with a case analysis on the last step as $H'; \overline{T}\, \overline{T'}\, T$ is obtained by repeated application of (D-SCHEDULE). By (WF-CONFIGURATION) and $active(T)$, we have $T \equiv \rho\, S\langle \mathsf{m}\, F\ \ \mathsf{s}\rangle$, $F(\mathsf{this}) = r_{\mathsf{this}}$, $H(r_{\mathsf{this}}) = \mathsf{C_{this}}|\omega|(\overline{r})$ and $mbody(\mathsf{C_{this}.m}) = (\overline{\mathsf{x}}; \overline{\tau_{\mathsf{z}}\, \mathsf{z}}; \mathsf{s_m}; \mathsf{return\, y})$ and $typeof(\mathsf{C_{this}.m}) = \overline{\tau_{\mathsf{m}}} \rightarrow \tau_{\mathsf{m}}$. By (WF-CONFIGURATION), $\vdash CT$ implies that all methods are well-typed and in particular there is an E such that $E \vdash \mathsf{s_m}$.

Case (D-RETURN):

1. $T \equiv \rho\, S'\langle \mathsf{m'}\, F'\ \ \mathsf{x = y'.m(\overline{z}); s'}\rangle\langle \mathsf{m}\, F\ \ \mathsf{return\ y}\rangle$ by (D-RETURN).
2. $\langle \mathsf{m'}\, F'\ \ \mathsf{x = y'.m(\overline{z}); s'}\rangle$ is WF in H by (WF-FRAME).
3. $E(\mathsf{y}) = \tau_{\mathsf{m}}$ by (T-METHOD).
4. $F(\mathsf{y}) = r_{\mathsf{y}}$ and $r_{\mathsf{y}} <:_H^{r_{\mathsf{this}}} \tau_{\mathsf{m}}$ by (WF-FRAME).
5. $T' \equiv \rho\, S'\langle \mathsf{m'}\, F'[\mathsf{x} \mapsto r_{\mathsf{y}}]\ \ \mathsf{s'}\rangle$ by (D-RETURN).
6. $F'(\mathsf{this}) = r'_{\mathsf{this}}$, $H(r'_{\mathsf{this}}) = \mathsf{C}|\omega'|(\overline{r'})$, $mbody(\mathsf{C.m'}) = (\overline{\mathsf{x_{m'}}}; \overline{\tau_{\mathsf{m'}}\, \mathsf{z_{m'}}};\ \mathsf{s_{m'}}; \mathsf{return\, y'})$, and $E(\mathsf{x}) = \tau_{\mathsf{x}}$ and $E(\mathsf{y'}) = \tau_{\mathsf{y'}}$ by (WF-CONFIGURATION).
7. $\tau_{\mathsf{x}} = adapt(\tau_{\mathsf{m}}, \tau_{\mathsf{y'}})$ by (T-CALL).
8. Show that $r_{\mathsf{y}} <:_H^{r'_{\mathsf{this}}} \tau_{\mathsf{x}}$, by case analysis on r_{y}.
   8.1. If $r_{\mathsf{y}} = \mathsf{null}$, then immediate by definition of run-time subtyping.
   8.2. If $r_{\mathsf{y}} \neq \mathsf{null}$. Let $H(r_{\mathsf{y}}) = \mathsf{C_y}|\omega_{\mathsf{y}}|(\overline{r''})$. We know that $r_{\mathsf{y}} <:_H^{r_{\mathsf{this}}} \tau_{\mathsf{m}}$.
      8.2.1. If $\tau_{\mathsf{m}} = \mathsf{D}$. Then $\tau_{\mathsf{x}} = \mathsf{D}$ by definition of *adapt* and $\mathsf{C_y} <: \mathsf{D}$, by the definition of run-time subtyping. $E(\mathsf{y}) = \tau_{\mathsf{m}}$ *is raw*, so *not* $internal_H(F(\mathsf{y}))$. Thus, $\mathsf{C_y}$ is not $\mathsf{internal}$. Therefore, by the definition of run-time subtyping, $r_{\mathsf{y}} <:_H^{r_{\mathsf{this'}}} \tau_{\mathsf{x}}$.
      8.2.2. If $\tau_{\mathsf{m}} = \mathsf{D|a = this.b|}$. $\mathsf{C_y} <: \mathsf{D}$, by the definition of run-time subtyping. Since τ_{m} *not raw*, we have $owner_H(r_{\mathsf{y}}) = owner_H(r_{\mathsf{this}})$. We have $E(\mathsf{y'}) = \tau_{\mathsf{y'}}$ *not raw*, for otherwise τ_{x} would be undefined, by the definition of *adapt*. We have $F'(\mathsf{y'}) <:_H^{r_{\mathsf{this'}}} \tau_{\mathsf{y'}}$, by (WF-FRAME).
      Therefore, $owner_H(F'(\mathsf{y'})) = owner_H(r'_{\mathsf{this}})$. But $F'(\mathsf{y'}) = F(\mathsf{this}) = r_{\mathsf{this}}$, by (T-CALL). So $owner_H(r_{\mathsf{this}}) = owner_H(r'_{\mathsf{this}})$. Thus, $owner_H(r_{\mathsf{y}}) = owner_H(r'_{\mathsf{this}})$. By the definition of run-time subtyping, $r_{\mathsf{y}} <:_H^{r_{\mathsf{this'}}} \tau_{\mathsf{x}}$.
9. T' is WF in H by (WF-THREAD-*).

Case (D-CAST):

1. $T \equiv \rho\, S\langle \mathsf{m}\, F\ \ \mathsf{x} = (\tau)\mathsf{y'}; \mathsf{s'}\rangle$ by (D-CAST).
2. $E(\mathsf{x}) = \tau_{\mathsf{x}}$ and $E(\mathsf{y'}) = \tau'_{\mathsf{y}}$ by (T-METHOD).
3. $F(\mathsf{y'}) = r'_{\mathsf{y}}$, and $r'_{\mathsf{y}} <:_H^{r_{\mathsf{this}}} \tau'_{\mathsf{y}}$ by (WF-FRAME).
4. $\tau_{\mathsf{x}} = \tau$ by (T-CAST-*).
5. Show that $r'_{\mathsf{y}} <:_H^{r_{\mathsf{this}}} \tau_{\mathsf{x}}$ by case analysis on τ_{x} and τ'_{y}:
   5.a. If $\tau'_{\mathsf{y}} = \mathsf{D}$ and $\tau_{\mathsf{x}} = \mathsf{C}$, then $\tau'_{\mathsf{y}} <: \tau_{\mathsf{x}}$ by (T-CAST-PLAIN). Since τ'_{y} *is raw*, then *not* $internal_H(r'_{\mathsf{y}})$. So τ'_{y} is not $\mathsf{internal}$. Therefore, by the definition of dynamic subtyping, $r'_{\mathsf{y}} <:_H^{r_{\mathsf{this}}} \tau_{\mathsf{x}}$.

5.b. If $\tau_{\mathsf{x}} = \mathsf{D}|\mathsf{a} = \mathsf{this.b}|$ and $\tau'_{\mathsf{y}} = \mathsf{C}|\mathsf{a}' = \mathsf{this.b}'|$ then $\mathsf{a} = \mathsf{a}'$, $\mathsf{b} = \mathsf{b}'$, and $\tau'_{\mathsf{y}} <: \tau_{\mathsf{x}}$ by (T-CAST-ASET). τ'_{y} *not raw*, and since $r'_{\mathsf{y}} <:^{r_{\mathsf{this}}}_{H} \tau'_{\mathsf{y}}$, we have $owner_H(r'_{\mathsf{y}}) = owner_H(r_{\mathsf{this}})$, by (WF-FRAME). Therefore $r'_{\mathsf{y}} <:^{r_{\mathsf{this}}}_{H} \tau_{\mathsf{x}}$, by the definition of run-time subtyping.

5.c. If $\tau_{\mathsf{x}} = \mathsf{D}$ and $\tau'_{\mathsf{y}} = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|$, then $\mathsf{C} = \mathsf{D}$ and C *not* internal by (T-CAST-OFF). Therefore $r'_{\mathsf{y}} <:^{r_{\mathsf{this}}}_{H} \tau_{\mathsf{x}}$, by the definition of run-time subtyping.

5.d. The case $\tau_{\mathsf{x}} = \mathsf{D}|\mathsf{a}{=}\mathsf{this.b}|$ and $\tau'_{\mathsf{y}} = \mathsf{C}$, has no type derivation.

6. $\langle \mathsf{m}\, F'[\mathsf{x} \mapsto r_{\mathsf{y}}]\ \ \mathsf{s}'\rangle$ is WF in H by (WF-FRAME) and (5).
7. $T' \equiv \rho\, S\langle \mathsf{m}\, F'[\mathsf{x} \mapsto r_{\mathsf{y}}]\ \ \mathsf{s}'\rangle$ is WF in H by (WF-THREAD-*) and (6).

Case (D-SKIP): Immediate.

Case (D-SELECT):
1. $T \equiv \rho\, S\langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{this.f}_i; \mathsf{s}'\rangle$ by (D-SELECT).
2. $typeof(\mathsf{C}_{\mathsf{this}}.\mathsf{f}_i) = \tau_{\mathsf{f}}$ by (T-SELECT).
3. $H(r.\mathsf{f}_i) = r'$ and $r' <:^{r_{\mathsf{this}}}_{H} \tau_{\mathsf{f}}$ by (WF-HEAP).
4. $E(\mathsf{x}) = \tau_{\mathsf{f}}$ by (T-SELECT).
5. $r' <:^{r_{\mathsf{this}}}_{H} E(\mathsf{x})$.
6. $\langle \mathsf{m}\, F[\mathsf{x} \mapsto r']\ \ \mathsf{s}'\rangle$ is WF in H by (5) and (WF-FRAME).
7. $T' \equiv \rho\, S\langle \mathsf{m}\, F[\mathsf{x} \mapsto r']\ \ \mathsf{s}'\rangle$ is WF in H by (6) and (WF-THREAD-*).

Case (D-UPDATE): Similar to case (D-SELECT).

Case (D-NEW-PLAIN):
1. $T \equiv \rho\, S\langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{new}\ \mathsf{C}(); \mathsf{s}'\rangle$ by (D-NEW-PLAIN).
2. r' is fresh, $v = \mathsf{C}|\epsilon|(\overline{\mathsf{null}})$, $H' = H[r' \mapsto v]$, $F' = F[\mathsf{x} \mapsto r']$ and *not* C *has* a by (D-NEW-PLAIN).
3. $E(\mathsf{x}) = \mathsf{C}$ and C *not* internal by (T-NEW-RAW).
4. $r' <:^{r_{\mathsf{this}}}_{H'} \mathsf{C}$ by definition of run-time subtyping.
5. $H'(r'.f) <:^{r'}_{H'} typeof(\mathsf{C.f})$
6. $\langle \mathsf{m}\, F[\mathsf{x} \mapsto r']\ \ \mathsf{s}'\rangle$ is WF in H' by (4) and (WF-FRAME).
7. $T' \equiv \rho\, S\langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s}'\rangle$ is WF in H' by (6) and (WF-THREAD-*).
8. $H' = H[r' \mapsto v]$ is WF in H' by (5) and (WF-HEAP).

Case (D-NEW-SELF):
1. $T \equiv \rho\, S\langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{new}\ \mathsf{C}(); \mathsf{s}'\rangle$ by (D-NEW-SELF).
2. r' is fresh, $v = \mathsf{C}|r'|(\overline{\mathsf{null}})$, $H' = H[r' \mapsto v]$, $F' = F[\mathsf{x} \mapsto r']$, and C *has* a by (D-NEW-SELF).
3. $E(\mathsf{x}) = \mathsf{C}$ and C *not* internal by (T-NEW-RAW).
4. $r' <:^{r_{\mathsf{this}}}_{H'} \mathsf{C}$ by definition of run-time subtyping.
5. $H'(r'.f) <:^{r'}_{H'} typeof(\mathsf{C.f})$
6. $\langle \mathsf{m}\, F[\mathsf{x} \mapsto r'\ \ \mathsf{s}'\rangle$ is WF in H' by (4) and (WF-FRAME).
7. $T' \equiv \rho\, S\langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s}'\rangle$ is WF in H' by (6) and (WF-THREAD-*).
8. $H' = H[r' \mapsto v]$ is WF in H' by (5) and (WF-HEAP).

Case (D-NEW-ALIAS):
1. $T \equiv \rho\, S\langle \mathsf{m}\, F\ \ \mathsf{x} = \mathsf{new}\ \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|(); \mathsf{s}'\rangle$ by (D-NEW-ALIAS).
2. $H(F(\mathsf{this})) = \mathsf{D}|r''{=}\mathsf{this}.\overline{r}|$, r' is fresh, $v = \mathsf{C}|r''|(\overline{\mathsf{null}})$, $H' = H[r' \mapsto v]$ and C *has* a by (D-NEW-ALIAS). Let $F' = F[\mathsf{x} \mapsto r']$.
3. $owner_H(r') = owner_H(r_{\mathsf{this}})$, by (2).
4. $r' <:^{r_{\mathsf{this}}}_{H'} \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|$ by definition of run-time subtyping.
5. $E(\mathsf{x}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|$ by (T-NEW-ASET).
6. $F'(\mathsf{x}) <:^{r_{\mathsf{this}}}_{H'} E(\mathsf{x})$ by (4) and (5).
7. $\langle \mathsf{m}\, F[\mathsf{x} \mapsto r']\ \ \mathsf{s}'\rangle$ is WF in H' by (6) (WF-FRAME).
8. $T' \equiv \rho\, S\langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s}'\rangle$ is WF in H' by (7) (WF-THREAD-*).
9. $H'(r'.\mathsf{f}) <:^{r'}_{H'} typeof(\mathsf{C.f})$, by the definition of run-time subtyping.

10. $H' = H[r' \mapsto v]$ is WF in H' by (9) and (WF-HEAP).

Case (D-CALL):

1. $T \equiv \rho\, S\langle m'\, F \;\; \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}'\rangle$ by (D-CALL).
2. $F(\mathsf{y}) = r'$, $F(\overline{\mathsf{z}}) = \overline{r}$, $H(r') = \mathsf{C}|\omega|(\overline{r'})$, $mbody(\mathsf{C.m}) = (\overline{\mathsf{x}'}; \overline{\tau_\mathsf{y}\, \mathsf{y}}; \mathsf{s}''; \mathsf{return}\ \mathsf{y}')$, $F' \equiv [\overline{\mathsf{y} \mapsto \mathsf{null}}][\overline{\mathsf{x}' \mapsto r}][\mathsf{this} \mapsto r']$, and $S' \equiv S\, \langle \mathsf{m}'\, F \;\; \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle\langle \mathsf{m}\, F' \;\; \mathsf{s}'; \mathsf{return}\ \mathsf{y}'\rangle$ by (D-CALL).
3. $typeof(\mathsf{C.m}) = \overline{\tau} \to \tau$, $E(\mathsf{y}) = \tau_\mathsf{y}$, $E(\overline{\mathsf{z}}) = \overline{\tau_\mathsf{z}}$, $\overline{\tau_\mathsf{z}} = adapt(\overline{\tau}, \tau_\mathsf{y})$, $\tau_\mathsf{x} = adapt(\tau, \tau_\mathsf{y})$, $E(\mathsf{x}) = \tau_\mathsf{x}$ by (T-CALL).
4. $\overline{r <:_H^{r_\mathsf{this}} \tau_\mathsf{z}}$ by (WF-FRAME).
5. Show $\overline{r <:_H^{r'} \tau}$. Consider r_i, show $r_i <:_H^{r'} \tau_i$, by case analysis on τ_i:
   5.a. If τ_i ***is raw***. We have $\tau_{\mathsf{z}_i} = adapt(\tau_i, \tau_\mathsf{y})$, so $\tau_{\mathsf{z}_i} = \tau_i$. $r_i <:_H^{r_\mathsf{this}} \tau_i$, by (4). So *not* $internal_H(r_i)$. Therefore $r_i <:_H^{r'} \tau_i$ by the definition of run-time subtyping.
   5.b. If τ_i ***not raw***. We have $\tau_{\mathsf{z}_i} = adapt(\tau_i, \tau_\mathsf{y})$. τ_y ***not raw*** for otherwise, ***adapt*** would be undefined, and τ_{z_i} ***not raw***. $owner_H(r_i) = owner_H(r_\mathsf{this})$, since $r_i <:_H^{r_\mathsf{this}} \tau_{\mathsf{z}_i}$. $owner_H(r') = owner_H(r_\mathsf{this})$, since $r' <:_H^{r_\mathsf{this}} \tau_\mathsf{y}$. So $owner_H(r_i) = owner_H(r')$. Therefore $r_i <:_H^{r'} \tau_i$ by the definition of run-time subtyping.
6. Show $r' <:_H^{F'(\mathsf{this})} E(\mathsf{this})$, by case analysis on C:
   6.a. If *not* C *has* a. Then $E(\mathsf{this})$ ***is raw***. C *not* $\mathsf{internal}$, by (T-CLASS). So $r' <:_H^{F'(\mathsf{this})} E(\mathsf{this})$, by the definition of run-time subtyping.
   6.b. If C *has* a. Then $E(\mathsf{this})$ ***not raw***. We have $\omega \neq \epsilon$ by (WF-HEAP). So $owner_H(r') \neq \epsilon$, and $r' <:_H^{F'(\mathsf{this})} E(\mathsf{this})$, by the definition of run-time subtyping.
7. $\langle \mathsf{m}\, F' \;\; \mathsf{s}'; \mathsf{return}\ \mathsf{y}'\rangle$ is WF in H, by (5) and (6).
8. Show $\exists\langle \mathsf{m}''\, F'' \;\; s''\rangle \in S'$ such that: $owner_H(F''(\mathsf{this})) = owner_H(F'(\mathsf{this}))$ and *not* $internal_H(F''(\mathsf{this}))$, by case analysis on r':
   8.a. If *not* $internal_H(r')$. Immediate, $\langle \mathsf{m}''\, F'' \;\; s''\rangle$ is $\langle \mathsf{m}\, F' \;\; \mathsf{s}'; \mathsf{return}\ \mathsf{y}'\rangle$.
   8.b. If $internal_H(r')$. Then τ_y ***not raw***. $owner_H(r') = owner_H(r_\mathsf{this})$, since $r' <:_H^{r_\mathsf{this}} \tau_\mathsf{y}$. We know that $\rho\, S\langle m'\, F \;\; \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}'\rangle$ is WF in H. So $\exists\langle \mathsf{m}''\, F'' \;\; s''\rangle \in S\langle m'\, F \;\; \mathsf{x} = \mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}'\rangle$ such that $owner_H(F''(\mathsf{this})) = owner_H(r_\mathsf{this}) = owner_H(r')$ and
   not $internal_H(F''(\mathsf{this}))$.
9. $\rho S'$ is WF by (7), (8), and (WF-THREAD-*).

Case (D-CALL-NPE): Immediate by (WF-NPE-THREAD).

□

Progress requires that if there exists an active thread in a well-formed configuration, this thread should be allowed to make a step.

THEOREM 4.3. *Progress. If* $H; \overline{T}\, T\, \overline{T'}$ *is WF and* $active(T)$*, then* $H; \overline{T}\, T\, \overline{T'} \xrightarrow{\ell}_\rho H'; \overline{T}\, \overline{T'}\, T'$.

PROOF. We obtain $H'; \overline{T}\, \overline{T'}\, T$ by repeated application of (D-SCHEDULE). We proceed by structural induction on s when $T \equiv \rho\, S\langle \mathsf{m}\, F \;\; \mathsf{s}\rangle$. By (WF-CONFIGURATION) and $active(T)$, $H(F(\mathsf{this})) = \mathsf{C}|\omega|(\overline{r})$ and $mbody(\mathsf{C.m}) = (\overline{\mathsf{x_m}}; \overline{\tau_\mathsf{m}\, \mathsf{z_m}}; \mathsf{s_m}; \mathsf{return}\, \mathsf{y})$. By (WF-CONFIGURATION), $\vdash CT$ implies all methods are well-typed and there is a E such that $E \vdash \mathsf{s_m}$.

Case [$s \equiv \mathsf{return}\ \mathsf{y}$]:

(a) By $\vdash CT$ and (WF-THREAD), $E(\mathsf{y}) = \tau_\mathsf{y}$, $F(\mathsf{y}) = r_\mathsf{y}$, $F(\mathsf{this}) = r$.
(b) By $active(T)$ and (WF-THREAD), $S = S'\langle \mathsf{m}'\, F' \;\; \mathsf{x} = \mathsf{y}'.\mathsf{m}(\overline{\mathsf{z}}); \mathsf{s}'\rangle$.

(c) By b) we can apply (D-RETURN) to obtain $H; \overline{T}\,\overline{T'}\,\rho\, S'\langle \mathsf{m'}\ F'\ \ \mathsf{x} = \mathsf{y'}.\mathsf{m}(\overline{\mathsf{z}}); \mathsf{s'}\rangle$.

Case $[\mathsf{s} \equiv \mathsf{s'}; \mathsf{s''}]$: Follows immediately by the induction hypothesis.

Case $[\mathsf{s} \equiv \mathsf{skip}; \mathsf{s'}]$:

(a) By (D-SKIP) we obtain $H; \overline{T}\,\overline{T'}\,\rho\, S\langle \mathsf{m}\ F\ \ \mathsf{s'}\rangle$.

Case $[\mathsf{s} \equiv \mathsf{x} = \mathsf{y}.\mathsf{f}_i; \mathsf{s'}]$:

(a) By $\vdash CT$ and (WF-THREAD), $E(\mathsf{y}) = \tau_{\mathsf{y}}$, $F(\mathsf{y}) = r_{\mathsf{y}}$, $F(\mathsf{this}) = r$.

(b) By a) either $r_{\mathsf{y}} = \mathsf{null}$ or $H(r_{\mathsf{y}}) = \mathsf{D}|\omega'|(\overline{r'})$.

(c) By b) if $r_{\mathsf{y}} = \mathsf{null}$ then by application of (D-SELECT-NPE) we obtain $H; \overline{T}\,\overline{T'}\,\rho\, \mathsf{NPE}$.

(d) By b) if $H(r_{\mathsf{y}}) = \mathsf{D}|\omega'|(\overline{r'})$, by (T-SELECT) and (WF-HEAP) there is a $r_i \in \overline{r'}$ corresponding to f_i.

(e) By d) and (D-SELECT) we obtain $H'; \overline{T}\,\overline{T'}\,\rho\, S\langle \mathsf{m}\ F\ \ \mathsf{s'}\rangle$.

Case $[\mathsf{s} \equiv \mathsf{x}.\mathsf{f}_i = \mathsf{y}; \mathsf{s'}]$: Similar to the previous case.

Case $[\mathsf{s} \equiv \mathsf{y} = (\tau)\mathsf{x}; \mathsf{s'}]$: Immediate by application of (D-CAST).

Case $[\mathsf{s} \equiv \mathsf{x} = \mathsf{new}\ \tau(); \mathsf{s'}]$:

(a) Either $\tau \equiv \mathsf{D}$ or $\tau \equiv \mathsf{D}|\mathsf{a}{=}\mathsf{this}.\mathsf{b}|$.

(b) If $\tau \equiv \mathsf{D}|\mathsf{a} = \mathsf{this}.\mathsf{b}|$, then by (T-NEW-ASET) C *has* a and $E(\mathit{this})$ *has* b, recall that $H(F(\mathsf{this})) = \mathsf{C}|\omega|(\overline{r})$, then by (D-NEW-ALIAS) we obtain $H[r \mapsto \mathsf{D}|\omega|(\overline{\mathsf{null}})]; \overline{T}\,\overline{T'}\,\rho\,\langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s'}\rangle$ with r fresh.

(c) If $\tau \equiv \mathsf{D}$, then if D *has* a, by (D-NEW-SELF) we obtain $H[r \mapsto \mathsf{D}|r|(\overline{\mathsf{null}})]; \overline{T}\,\overline{T'}\,\rho\,\langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s'}\rangle$ with r fresh, otherwise by (D-NEW-PLAIN) we obtain $H[r \mapsto \mathsf{D}|\epsilon|(\overline{\mathsf{null}})]; \overline{T}\,\overline{T'}\,\rho\,\langle \mathsf{m}\, F[\mathsf{x} \mapsto r]\ \ \mathsf{s'}\rangle$.

Case $[s \equiv \mathsf{x} = \mathsf{y}.\mathsf{m'}(\overline{\mathsf{z}}); s']$:

(a) By (WF-THREAD), (WF-HEAP) and application of (D-CALL) we obtain $H; \overline{T}\,\overline{T'}\,\rho\, S\langle \mathsf{m}\ F\ \ s\rangle\langle m'\ F'\ \ s'\rangle$.

□

4.5. Concurrency Control

The AJ semantics is purposefully silent about synchronization to allow for different concurrency-control strategies. Our implementation uses mutual exclusion locks, our previous work used read-write locks, and a transactional implementation would be another possibility. The execution of a program can be characterized by a trace t which is a sequence of events $e_1 \ldots e_n$ performed by individual threads. For any implementation of AJ, we define the concurrency-control policy as a predicate over traces. We say that any trace accepted by an implementation is *well-formed*. The current implementation disallows multiple invocations of methods on objects having the same owner to execute concurrently by associating mutual exclusion locks to atomic set instances. We formalize this with the following definition of valid event. Let an event e be a tuple $(H, \overline{T}, \ell, \rho)$ consisting of a configuration, an action label and a thread id. We say that an event is valid if it has any action label other than a method call. An event with a method call on an object of an internal class is valid. For calls to non-internal classes, an event is valid if there are no outstanding method calls of objects with the same owner in other threads.

Definition 4.4. An event $e = (H, \overline{T}, \ell, \rho)$ is *valid* if and only if:
when $\ell = \rightarrow r.\mathsf{m}$, $H(r) = \mathsf{C}|r'|(\overline{r})$ and C *not* internal
then $\nexists \rho' S \in \overline{T}.\rho' \neq \rho$ and $\langle \mathsf{m}\, F\ \ \mathsf{s}\rangle \in S$ and $H(F(\mathsf{this})) = \mathsf{D}|r'|(\overline{\mathsf{z}})$.

In our implementation, a well-formed trace is a trace in which every event is valid and every configuration is WF. This property, enforced by the AJ run-time system, is not sufficient in itself to prevent data races. The type system provides the additional guarantee that all objects belonging to an atomic set are accessed only through methods that are units of work for the atomic set.

4.6. Atomic-Set Serializability

Serializability of atomic set operations follows from the above restriction to valid traces (mutual exclusion of methods of non-internal classes operating on the same atomic set) and the fact that all fields labeled $\mathsf{atomic(a)}$, including those of internal classes, are accessed within a method of a non-internal class operating on that atomic set. Given a well-formed trace t and an event e in t, $\textit{aset}_t(e)$ gives the owner atomic set accessed by e, if any.

$$\textit{aset}_t(e) = \begin{cases} r' & \text{if } e = (H, \overline{T}, \ell, \rho) \wedge \ell \in \{\uparrow r.\mathsf{f}, \downarrow r.\mathsf{f}\} \\ & \wedge\ H(r) = \mathsf{C}|r'|(\overline{r}) \wedge \mathsf{C.f}\ \mathit{is}\ \mathsf{atomic} \\ \\ \epsilon & \text{otherwise.} \end{cases}$$

We introduce *unit of work identifiers*, ranged over by meta variable u, in a trace t as follows. We consider the projection of t onto each thread ρ, which is a succession of events from the same thread. By considering method calls and returns ($\rightarrow r.\mathsf{m}$, $\leftarrow r.\mathsf{m}$), we determine where units of work start and end. We assign each unit of work a unique identifier u, and update all frames in the trace t to reflect not only the method name, but also the unit of work identifier u, as follows: $\langle \mathsf{m}\, u\, F\ \ s\rangle$. Given a well-formed trace t, and an event e, $\textit{uow}_t(e)$ is the unit of work to which e belongs. $\textit{uow}_t(e)$ is computed by examining the call stack of the thread that performs e, finding the *first* frame on the stack with a method on an object having the same owner as $\textit{aset}_t(e)$, declared in a non-internal class, and returning the unit of work identifier corresponding to this method.

$$\textit{uow}_t(e) = \begin{cases} u & \text{if } e = (H, \overline{T}\rho S, \ell, \rho) \wedge \exists \langle \mathsf{m}\, u\, F\ \ \mathsf{s}\rangle \in S\ \mathit{s.t.} \\ & \quad \mathit{owner}_H(F(\mathsf{this})) = \textit{aset}_t(e) \\ & \quad \wedge\ \mathit{not\ internal}_H(F(\mathsf{this})) \\ & \quad \wedge\ \nexists \langle \mathsf{m}'\, u'\, F'\ \ \mathsf{s}'\rangle \ldots \langle \mathsf{m}\, u\, F\ \ \mathsf{s}\rangle \in S \\ & \qquad \mathit{s.t.}\ \mathit{owner}_H(F'(\mathsf{this})) = \textit{aset}_t(e) \\ & \qquad \wedge\ \mathit{not\ internal}_H(F'(\mathsf{this})) \\ \\ \perp & \text{otherwise} \end{cases}$$

Lemma 1. If $e = (H, \overline{T}, \ell, \rho)$ is an event in a well-formed trace t and $\textit{aset}_t(e) \neq \epsilon$, then $\textit{uow}_t(e) \neq \perp$.

PROOF. Let $e = (H, \overline{T}\rho S\langle \mathsf{m}'\, F'\ \ \mathsf{s}'\rangle, \ell, \rho)$. Since $\textit{aset}_t(e) = r' \neq \epsilon$, we have $\ell \in \{\uparrow r.\mathsf{f}, \downarrow r.\mathsf{f}\}$, $H(r) = \mathsf{C}|r'|(\overline{r})$, and $\mathsf{C.f}\ \mathit{is}\ \mathsf{atomic}$. Fields can only be accessed from this, so $r = F'(\mathsf{this})$. By (WF-THREAD), we know that there exists a frame $\langle \mathsf{m}\, F\ \ s\rangle$ in S such that $\mathit{owner}_H(F(\mathsf{this})) = \mathit{owner}_H(F'(\mathsf{this})) = \textit{aset}_t(e)$, and $\mathit{not\ internal}_H(F(\mathsf{this}))$. Therefore, $\textit{uow}_t(e) \neq \perp$. □

The *events of a unit of work* u in a trace t are all the events e in t such that $\textit{uow}_t(e) = u$. Given a well-formed trace t and an atomic set r, we define the set of *units of work corresponding to* r as the set that contains $\textit{uow}_t(e)$ for each e in t such that $\textit{aset}_t(e) = r$. By Lemma 1, we know that $\textit{uow}_t(e)$ is well-defined for an event e such that $\textit{aset}_t(e) \neq \epsilon$, meaning each access to a location in an atomic set is performed within a unit of work corresponding to that atomic set. Since valid traces provide mutual exclusion of units of work, we obtain atomic-set serializability.

THEOREM 4.5. *Atomic-Set Serializability. Given a well-formed trace t and an atomic set r, the events of each of the units of work corresponding to r happen serially.*

PROOF. By contradiction. Assume that t contains 3 events e, e', and e'' in this order, such that: $aset_t(e) = aset_t(e') = aset_t(e'') = r$, and $uow_t(e) = uow_t(e'') \neq uow_t(e')$. Assume that e' is performed by a different thread than e and e'', and that $e' = (H', \overline{T}, \rightarrow r.\mathsf{m}, \rho)$. Since trace t is well-formed, we know that e' is valid. By the definition of valid event, there is no other thread in the configuration of e' that has an invocation of a method in the same atomic set on its call stack. However, since e and e'' belong to the same unit of work, this means when e' occurs, unit of work $uow_t(e)$ has not yet ended. Therefore, e' is not valid, which is a contradiction. Therefore no invocation by another thread of a method on atomic set r may be interleaved between e and e''. Since all accesses to r happen inside a method operating on r, no other event accessing r by another thread may be interleaved. So units of work corresponding to r happen serially. □

4.7. Adding unitfor

AJ provides a feature to dynamically expand a unit of work to multiple atomic sets. This is done by annotation of method arguments with the unitfor modifier. At run time, the locks of all the named atomic sets are acquired and the method is serializable with respect to these atomic sets. Consider a hypothetical method addAll2() in the LinkedList class:

```
class LinkedList extends AbsList {
  ...
  void addAll2(unitfor(a) AbsList l1,unitfor(a) AbsList l2) {
    ...
```

The programmer specified that the method is a unit of work for the receiver of the call as well as for both arguments. This is the only way to ensure that neither the receiver nor the arguments are modified concurrently. Semantically, the method invocation will acquire all locks atomically. (Our implementation uses a lock ordering protocol to prevent deadlocks.) We now sketch the changes to the formalism to support unitfor. First, the syntax of the calculus is extended with optional unitfor annotations on method arguments.

$$\begin{array}{lcl} u & ::= & \mathsf{unitfor\ (a)} \mid \epsilon \\ md & ::= & \tau\ \mathsf{m}\ (\overline{u\ \tau\ \mathsf{x}})\ \{\overline{\tau\ \mathsf{z}};\mathsf{s;return\ y}\} \end{array}$$

Next, the type checking rule for methods is adapted. As atomic sets are inherited, we deem it natural to enforce the constraint that subclasses preserve the synchronization behavior of their parent. The new type rule assumes that the *override* predicate checks that the unitfor specifications, $\overline{u}$ match those of the method declaration D.m.

$$\frac{\begin{array}{c}E \equiv \overline{\mathsf{x} : \tau_{\mathsf{x}}}, \overline{\mathsf{z} : \tau_{\mathsf{z}}}, \mathsf{this} : \tau_{\mathsf{this}} \quad E\ \vdash\ \mathsf{s; return\, y} \quad E(\mathsf{y}) = \tau \quad \mathsf{C\ extends\ D} \\ (\mathit{if}\ \mathsf{C}\ \mathit{has}\ \mathsf{a}\ \ \mathit{then}\ \ \tau_{\mathsf{this}} \equiv \mathsf{C}|\mathsf{a}{=}\mathsf{this.a}|\ \ \mathit{else}\ \ \tau_{\mathsf{this}} \equiv \mathsf{C}) \quad \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{u\ \tau_{\mathsf{x}}} \rightarrow \tau)\end{array}}{\tau\, \mathsf{m}(\overline{u\ \tau_{\mathsf{x}}\, \mathsf{x}})\{\overline{\tau_{\mathsf{z}}\, \mathsf{z}};\ \mathsf{s; return\, y}\} \quad \text{OK in C}}\ \text{(T-METHOD')}$$

The next change is in the dynamic semantics. Whereas before, it was sufficient to record method calls as pairs of receiver object and method name, $\xrightarrow{\rightarrow r.\mathsf{m}}_{\rho}$, we now also record the subset of the method's arguments that have associated unitfor annotations, $\xrightarrow{\rightarrow r.\mathsf{m}\ \overline{r_u}}_{\rho}$. The new rule for a method call relies on predicate $units(\mathsf{C}, \mathsf{m}, \overline{r})$ to return the subset of the arguments $\overline{r_u}$ corresponding to unitfor parameters. Call frames are extended from triples $\langle \mathsf{m}\, F\ \ \mathsf{s}\rangle$ to quadruples $\langle \mathsf{m}\, F\ \overline{r_u}\ \ \mathsf{s}\rangle$ by addition of the unitfor arguments.

$$\frac{\begin{array}{c} F(\mathsf{y}) = r \quad F(\overline{\mathsf{z}}) = \overline{r} \qquad H(r) = \mathsf{C}|\omega|(\overline{r'}) \qquad mbody(\mathsf{C.m}) = (\overline{\tau_{\mathsf{x}}\,\mathsf{x}'};\ \overline{\tau_{\mathsf{y}}\,\mathsf{y}}; \mathsf{s}'; \mathsf{return}\ \mathsf{y}') \\ F' \equiv [\overline{\mathsf{y} \mapsto \mathsf{null}}][\overline{\mathsf{x}' \mapsto r}][\mathsf{this} \mapsto r] \qquad \overline{r_u} = units(\mathsf{C}, \mathsf{m}, \overline{r}) \\ S' \equiv S\,\langle \mathsf{m}'\, F\ \overline{r'_u}\ \ \mathsf{x}{=}\mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle\langle \mathsf{m}\, F'\ \overline{r_u}\ \ \mathsf{s}'; \mathsf{return}\ \mathsf{y}'\rangle \end{array}}{H; \overline{T}\,\rho\, S\,\langle \mathsf{m}'\, F\ \overline{r'_u}\ \ \mathsf{x}{=}\mathsf{y.m}(\overline{\mathsf{z}}); \mathsf{s}\rangle \xrightarrow{\to r.\mathsf{m}\ \overline{r_u}}_{\rho} H; \overline{T}\,\rho\, S'}\ \text{(D-CALL')}$$

No other changes are required to the semantics. The definitions of well-formed configurations and the proofs are unchanged. The definition of *valid* events must be adjusted to treat the extra arguments on call events as locks requiring mutual exclusion. This is achieved by ensuring that the set of locks required by an event e are disjoint from those of any stack frame in the configuration.

Definition 4.6. An event $e = (H, \overline{T}, \ell, \rho)$ is *valid* if and only if:
when $\ell = \!\!\to r.\mathsf{m}\ \overline{r_u}$, $H(r) = \mathsf{C}|r'|(\overline{r})$
then $\nexists\, \rho' S \in \overline{T}.\rho' \neq \rho$ and $\langle \mathsf{m}\, F\ \overline{r'_u}\ \ \mathsf{s}\rangle \in S$ and $\big(H(F(\mathsf{this})) \cup \overline{r_u}\big) \cap \big(\mathsf{D}|r'|(\overline{\mathsf{z}}) \cup \overline{r'_u}\big) = \emptyset$.

The treatment of concurrency must be adjusted slightly to account for the fact that methods are protected by multiple atomic sets. This affects the definition of *uow* and the statement of the theorem. The result follows as expected.

5. IMPLEMENTATION: TRANSLATING AJ TO JAVA

We implemented a proof-of-concept AJ-to-Java compiler as an Eclipse refactoring that rewrites the original source into a new project that holds the transformed code. The type checker assumes that data-centric synchronization annotations are given as Java comments. It parses these annotations and enforces the type rules of Section 4. Type errors are reported using markers in the Eclipse editor. The compiler uses standard Java synchronized blocks to enforce exclusion for each atomic set. A limitation of our prototype is that it supports only one atomic set per class. Furthermore, it does not handle generics and nested classes. We emphasize that this is not a limitation of the approach, but an engineering tradeoff. With Eclipse's rudimentary support for AST manipulation handling those features would entail a considerable effort. Therefore, when these features are encountered in Java code to be used in AJ, we perform manual refactorings to side-step the problem. Generics are eliminated by removing type parameters and replacing occurrences of these type parameters with type Object. Nested classes are dealt with in two steps. First, any non-static nested class is changed into a static nested class by introducing an explicit pointer to the surrounding object. Then, the nested classes are changed into top-level classes.

The prototype implements a four-step transformation that ensures that each non-private method of a non-internal class acquires the locks for all atomic sets for which it is a unit of work. We also experimented with an alternative implementation, based on reentrant locks from java.util.concurrent but found the performance inferior to the current implementation that is based on synchronized blocks.

5.1. Transformation steps

5.1.1. Create lock fields. The compiler generates a lock field \$lock_S in any class C that declares an atomic set S. Atomic sets declared in super-interfaces of C will have a lock field in C unless that same atomic set is present in C's superclass. For each lock field, an accessor method getLockForS() is created.

5.1.2. Transform constructors. Constructors of classes with atomic sets are transformed to take additional parameters that are the lock objects to use. For classes that declare atomic sets, the constructors assign these parameters to the lock fields; for classes that inherit atomic sets, these lock objects are passed to superclass constructors.

5.1.3. Transform object allocations to set locks. For objects not involved in alias relationships, new statements are transformed by passing a fresh lock object to the constructor. For objects in an alias relationship, the lock to use is read from the owner by calling the getLockForS() accessor method and passed to the constructor to initialize the lock field.

5.1.4. Transform units of work to acquire all needed locks. This involves taking the lock of the atomic set of the declaring class and the locks for the atomic sets of any unitfor parameters. If only a single lock is required, a single synchronized block suffices. However, when multiple locks are needed, they must be acquired without inducing unnecessary deadlock. This is accomplished by introducing an ordering: each lock object is given an id when allocated, and locks are acquired in order of increasing id. There is a minor complication here: when the type of the argument is too general to denote an atomic set unambiguously, a unitfor must be used that omits the name of the atomic set (this situation arises, e.g., for the argument of equals() methods, see section 6.2). To this end, each class with atomic sets implements an interface Atomic, which declares a method getLock() that returns the lock for its atomic set.

```
class F {
  atomicset f;
  B myB |b=this.f|;
  atomic(f) long fCounter;
  ...
  void foo(unitfor(b) B b1, B b2) {
    fCounter++;
    b1.bar(myB); //no-lock version
    b2.bar(myB); //locking version
  }
}
```

```
class B {
  atomicset b;
  atomic(b) long bCounter;
  ...
  void bar(unitfor(b) B that) {
    this.inc(); //no-lock version
    that.inc(); //no-lock version
  }
  void inc(){ bCounter++; }
}
```

Fig. 15: Example where the compiler can determine that locks need not be reacquired.

A few straightforward optimizations were implemented. If the compiler can determine that all members of an atomic set accessed in a method and in any methods it may call are final, then it will not introduce locking code. Furthermore, all transformed methods have two versions, one with locking code and one without; when the compiler can determine that all needed locks are already held in a particular context, it will call the version that does not take locks. Consider the code in Figure 15 as an example. The compiler knows that while executing method foo(), locks for atomic sets this.f and b1.b are held. Furthermore, the aliasing annotation on myB indicates that myB.b and this.f are in fact the same lock. Therefore, the call b1.bar(myB) can be translated to use the version of bar that does not acquire any locks since all necessary locks are already held. The comments in the figure indicate whether the locking or non-locking version of a method will be called at each call site.

5.2. Translation Example

To illustrate the translation described above, we show the translated version of the code in Figure 15: class F in Figure 16 and class B in Figure 17.

```
1  public class F implements atomicsets.Atomic {
2    /*atomicset(f)*/
3    F(OrderedLock f) {
4      super();
5      $lock_f = f;
6    }
7
8    public final OrderedLock getLockForf() {
9      return $lock_f;
10   }
11
12   public final OrderedLock getLock() {
13     return this.getLockForf();
14   }
15
16   B myB /*b=this.f*/;
17   /*atomic(f)*/ long fCounter;
18
19   void foo_internal(/*unitfor(b)*/ B b1, B b2) {
20     fCounter++;
21     b1.bar_internal(myB); //no-lock version
22     b2.bar(myB); //locking version
23   }
24
25   void foo(B b1, B b2) {
26     OrderedLock l1 = null, l2 = null;
27     OrderedLock l3 = b1.getLockForb();
28     OrderedLock l4 = this.$lock_f;
29     if (l3.getIndex() > l4.getIndex()) {
30        l1 = l3; l2 = l4;
31     } else {
32        l1 = l4; l2 = l3;
33     }
34     synchronized (l1) {
35       synchronized (l2) {
36         fCounter++;
37         b1.bar_internal(myB); //no-lock version
38         b2.bar(myB); //locking version
39       }
40     }
41   }
42   protected final OrderedLock $lock_f;
43 }
```

Fig. 16: Translation for class F in Figure 15

(1) Create lock fields. The lock fields themselves are added to each class that declares an atomic set, as illustrated with the locks at line 42 in Figure 16 and 43 in Figure 17. Note that the locks are of type OrderedLock; we impose a global order on all locks allocated and we use this global order to ensure that we do not get spurious deadlocks from trying to acquire the same locks together in a different orders at different points in the code. The getLock() methods for the declared atomic sets are shown in lines 8-14 in Figure 16 and lines 4-10 in Figure 17.
(2) Transform constructors. The classes F and B each declare an atomic set, so their constructors take a parameter that denotes the lock object to use. In class F, the

constructor is on lines 3-6 and the assignment of the passed-in lock object is on line 5. Class B is similar.

(3) Transform object allocations to set locks. This example has no instances of object creation, but creations of these objects would receive additional lock objects as parameters.

(4) Transform units of work to acquire all needed locks. The method inc() of class B illustrates the simple case of a unit of work on a single atomic set, in this case b. Two versions of the code are generated. One version takes the one needed lock in a standard synchronized block; this is shown in lines 37-41 of Figure 17. The internal version (line 35) takes no lock and is used when the caller is known to have the lock already, for instance in the calls within bar() on lines 30, 31.

The method foo() in class F (lines 19-41 in Figure 16) illustrates some more translation issues. Note that myB is declared aliased with the F object, as indicated by the b=this.f aliasing annotation on line 16. This means that the referent shares the same lock as the F object itself, and hence calls on the referent's units of work require no additional locking. This is implemented by using the special internal version of such units of work that do not take locks, as shown on line 21 and line 37. The foo() method also declares the parameter b1 to be unitfor, meaning foo() is a unit of work for that object as well. This means that foo() must also take the lock for b1. The OrderedLock class allows the system to take locks in a global order so that multiple units of work trying in parallel to take multiple locks will not encounter spurious deadlocks. This locking code is shown on lines 26-35; the system determines which lock is earlier in the global order, and takes that lock first followed by the second needed lock.

6. EXTENDING AJ

This section presents extensions to the basic AJ programming model to support better integration with traditional Java programming idioms. In particular, Section 6.1 defines a notion of condition variables at the level of atomic sets, in order to support explicit synchronization between threads analogous to Java's wait() and notify(). Section 6.2 presents a generalized form of the unitfor construct, which is useful in cases where the name of an atomic set in an object is not known at compile time. Section 6.3 presents dynamic casts, a feature for converting a raw type into an aliased type (which requires a run-time check). In Section 6.4, we slightly extend the language to allow unitfor constructs to refer to final fields. Section 6.5 defines a fastread modifier on methods to achieve a relaxed synchronization policy that we found to be useful for achieving good performance in optimal solution search problems. Finally, Section 6.6 presents a notunitfor construct that programmers should use judiciously, to indicate that a given method is not a unit of work for the atomic set(s) in its declaring class.

6.1. Supporting Wait/Notify Synchronization

In Java, the Object class provides three additional methods for synchronizing the execution of multiple threads, wait(), notify() and notifyAll(). The Java monitor semantics requires that, in order for a thread to evaluate an expression such as obj.wait(), the thread must have acquired the receiver's lock. The call to wait() has the effect of releasing the lock associated with obj and suspending the thread. The thread is reactivated when some other thread calls obj.notify() or obj.notifyAll().

In AJ, Java's wait()/notify() mechanism cannot be used because this construct is not aware of the locks associated with atomic sets. For instance, consider the Count class in Figure 18(a), which declares an atomic set a and a method add(). Calling wait() within

```
public class B implements atomicsets.Atomic {
  B(OrderedLock b) {  super();   $lock_b = b;  }

  public final OrderedLock getLockForb() {
    return $lock_b;
  }

  public final OrderedLock getLock() {
    return this.getLockForb();
  }
  /*atomicset(b)*/
  /*atomic(b)*/ long bCounter;

  void bar_internal(/*unitfor(b)*/ B that) {
    this.inc_internal(); //no-lock version
    that.inc_internal(); //no-lock version
  }

  void bar(B that) {
    OrderedLock l1 = null, l2 = null;
    OrderedLock l3 = that.getLockForb();
    OrderedLock l4 = this.$lock_b;
    if (l3.getIndex() > l4.getIndex()) {
       l1 = l3; l2 = l4;
    } else {
       l1 = l4;  l2 = l3;
    }
    synchronized (l1) {
      synchronized (l2) {
        this.inc_internal(); //no-lock version
        that.inc_internal(); //no-lock version
      }
    }
  }
  void inc_internal(){ bCounter++; }

  void inc() {
    synchronized (this.$lock_b) {
      bCounter++;
    }
  }

  protected final OrderedLock $lock_b;
}
```

Fig. 17: Translation for class B in Figure 15

the body of the method is not allowed by Java semantics as the current thread does not hold the lock on this.[1]
However, some common Java programming idioms rely on wait() and notify() and it would be very difficult to do without this feature (in particular, some of the benchmark programs that we refactored into AJ rely on this feature). Therefore, we extend AJ with a special form of wait()/notify() which lets programmers write expressions such

[1] Our implementation does allow uses of Java's wait()/notify() that involve locks unrelated to atomic sets.

```
class Count {                    class Count {
    atomicset a;                     atomicset a;
    atomic (a) int cnt;              atomic (a) int cnt;
    void add(int x) {                void add(int x) {
        this.wait();                     this.a.wait();
        cnt =+ x;                        cnt =+ x;
    }                                }
}                                }
            (a)                              (b)
```

Fig. 18: **(a)** A problematic usage of wait()/notify() in AJ. **(b)** Example illustrating AJ's data-centric wait()/notify() construct.

as this.a.wait() where a is an atomic set in the receiver object. Figure 18(b) shows a revised version of the example that uses this construct. The semantics of this data-centric wait()/notify() construct are to release the lock(s) associated with the current unit of work and block the current thread. This effectively turns the method into *two* units of work, which are separated by the call to wait(). Thus, in the add() method in Figure 18(b), wait() would release the lock associated with atomic set a in the receiver. After notification, the thread will attempt to reacquire the same lock atomically. At present, our implementation supports wait()/notify() only in methods that are units of work for one atomic set.

6.2. Generic unitfor

Maintaining backwards compatibility with libraries is sometimes inconvenient as the signatures of common methods are too general. This is nicely illustrated by the equals(Object obj) method which does not expect an object with atomic sets. Of course, usually the argument obj is of the same type as the receiver and has the same atomic sets. Consider a Point class that has two mutable fields. To compare two objects of this class it is desirable to observe consistent states of the points. In Java this could be achieved by declaring the equals() method synchronized and acquiring the lock on the argument using a nested synchronized block.

```
class Point {
    int x,y;
    ...
    public synchronized boolean equals(Object o) {
        if (o==null || !(o instanceof Point)) return false;
        Point p = (Point)o;
        synchronized(p) { return x == p.getX() && y == p.getY(); }
    }
}
```

The above, of course, may result in a deadlock if two threads evaluate p.equals(q) and q.equals(p) in parallel. The equivalent solution with atomic sets requires an additional method, eq, with a unitfor argument to prevent concurrent modifications to the argument Point object.

```
class Point {
    atomicset a;
    atomic(a) int x,y;
    ...
    public boolean equals(Object o) {
        if (o==null || !(o instanceof Point)) return false;
        return eq((Point)o);
    }
    private boolean eq(unitfor(a) Point p) {
        return x == p.getX() && y == p.getY();
    }
}
```

Unfortunately, the AJ solution runs the same risks as the Java solution. A deadlock can occur when two points are compared in parallel. This situation arises because a lock on the receiver's atomic set is automatically acquired when equals() is called, and a second lock is acquired when eq() is called.

We propose a solution based on the notion of *generic* unitfor annotations. A generic unitfor annotation does not specify the name of the atomic set that has to be acquired. It has the semantics of atomically acquiring *all* atomic sets of the corresponding argument. If the argument is null or doesn't have atomic sets, nothing is done. The equals method can now be expressed more naturally.

```
    public boolean equals(unitfor Object o) {
        if (o==null || !(o instanceof Point)) return false;
        Point p = (Point) o;
        return x == p.getX() && y == p.getY();
    }
```

This solution does not run the risk of causing deadlocks as the locks on all atomic sets are acquired atomically. No changes to the type system are required.

6.3. Dynamic Casts

In order to support legacy code it is sometimes convenient to go from raw types to aliased types. The AJ type system allows casts from alias types to raw types, but not the other way around. Supporting dynamic casts to aliased types requires comparing types and atomic sets. To support this, we extend the the type system with one additional rule.

$$\frac{E(\mathsf{y}) = \mathsf{C}|\mathsf{a}{=}\mathsf{this.b}| \quad \mathsf{C}\ \mathit{has}\ \mathsf{a} \quad E(\mathsf{this})\ \mathit{has}\ \mathsf{b} \quad E(\mathsf{x}) = \mathsf{D} \quad \mathsf{C} <: \mathsf{D}}{E \vdash \mathsf{y} = (\mathsf{C}|\mathsf{a}{=}\mathsf{this.b}|)\mathsf{x}}\ \text{(T-DOWNCAST)}$$

Furthermore, the dynamic semantics must be extended with a downcast rule that performs a dynamic check on classes and compares the atomic set of the object being cast to the atomic set of the receiver.

$$\frac{H(F(\mathsf{this})) = \mathsf{C}|\omega|(\overline{r}, r_i, \overline{r'}) \qquad H(F(\mathsf{y})) = \mathsf{D}|\omega'|(\overline{r}', r_i', \overline{r''}) \qquad \mathsf{D} <: \mathsf{C} \qquad \omega = \omega'}{H; \overline{T}\,\rho\,S\,\langle \mathsf{m}\,F\ \ \mathsf{x}{=}(\mathsf{C}—a = this.b—)\mathsf{y}; \mathsf{s}\rangle \xrightarrow{\epsilon}_{\rho} H; \overline{T}\,\rho\,S\,\langle \mathsf{m}\,F[\mathsf{x} \mapsto F(\mathsf{y})]\ \ \mathsf{s}\rangle}\ \text{(D-CAST')}$$

As an example, consider an equals() method, which takes an Object as argument. In order to call this method it is necessary to convert the type of the argument to the general Object type, even if the only sensible value for equals() is one that matches the type of the receiver. Consider a Tree class with an equals() method. In this design the programmer chose coarse grained locking. This choice is manifested by the constraint that the left and right fields of a Tree instance must have the same atomic set a.

```
class Tree {
    atomicset a;
    atomic(a) Tree left|a=this.a|, right|a=this.a|;
    Tree|a=this.a| getLeft() { return left; }
    ...
    boolean equals(Object o) {
        if (!o instanceof Tree) return false;
        if (o == this) return true;
        ...
    }
    void setLeft(Tree t) { ... }
}
```

The body of the equals() method starts with the standard boilerplate Java idioms testing for null values and subtyping, and for reference equality. The ellipsis can be filled by the following code fragment:

```
Tree t = (Tree) o;
return left.equals(t.getLeft()) && right.equals(t.getRight());
```

Here the argument o is cast to the raw type Tree. This means that there is no guarantee that the object has the same atomic set. The call to t.getLeft() will have to acquire a lock. Of course if the argument has the same atomic set as the receiver, the lock is already held and it simply needs to be reentered.

With dynamic casts, the implementation could also include the following code:

```
Tree|a=this.a| t = (Tree|a=this.a|) o;
return left.equals(t.getLeft()) && right.equals(t.getRight());
```

In this case, since the AJ compiler knows that t is protected by the same atomic set, no additional lock needs to be acquired. Putting all this together the proper implementation of equals() would look as follows:

```
boolean equal(Object o) {
    if (o == null || !o instanceof Tree) return false;
    if (o == this) return true;
    if (o instanceof Tree|a=this.a|) {
        Tree|a=this.a| t = (Tree|a=this.a|) o;
        return left.equals(t.getLeft()) && right.equals(t.getRight());
    else { return eq((Tree)t) }
}
boolean eq(unitfor(a) Tree t) {
    return left.equals(t.getLeft()) && right.equals(t.getRight());
}
```

Another reason for having dynamic casts is to support assignments. Consider the setLeft(Tree) method. It takes a Tree object and should set the left field, but this is only permitted if the argument has the same atomic set as the receiver. The implementation of the methods would be:

```
void setLeft(Tree t) {
    if (t instanceof Tree|a=this.a|)
        left = (Tree|a=this.a|) o;
    else
        ... // error
}
```

In order to implement this feature, we rely on the fact that our implementation stores the lock associated with an atomic set in a field. We use these lock fields as a basis for comparisons. Every instance of a class that has an atomic set has a non-null value, and comparing the values of two fields will tell us if the atomic sets are the same. Internal classes can be treated specially as the type system does not allow upcasts to non-atomic set classes.

6.4. Generalized unitfor for Fields

In object-oriented code, it is natural for methods to manipulate fields of the object to which they belong. As such, it is sometimes useful to specify atomicity requirements on these fields. But the basic AJ programming model allows unitfor annotations to modify only method parameters. While this doesn't limit expressiveness, it leads to inelegant code when a method is a unit of work for some component object's atomic set; the method must call a helper method that accepts the field as a parameter. To avoid this, we extend our implementation to allow an additional form of unitfor annotations on methods. These unitfor method annotations indicate that the method is a unit of work for an atomic set of an object stored in a specified field. In order to avoid unsoundness arising from concurrent field updates, we require that fields specified in unitfor annotations be final. This restriction is conservative; a more permissive implementation could allow a non-final field as long as the field itself was part of an atomic set and the annotated method was also a unit of work for that atomic set.

As an example of using the generalized unitfor, we can write a transfer function for a linked account object that contains two bank account objects, each of which has an atomic set a, as follows:

```
class LinkedAccount {
  final Account checking, savings;
  ...
  void unitfor(checking.a) unitfor(savings.a) transferToChecking(int amt) {
    savings.withdraw(amt);
    checking.deposit(amt);
  }
}
```

Fig. 19: Generalized unitfor example.

To allow programmers maximum flexibility, we allow the unitfor annotations to specify atomic sets in fields of fields, fields of parameters, fields of fields of fields, and so on to an arbitrary depth. Again, to avoid unsoundness, each of the fields involved in naming the atomic set must be final or be part of an atomic set for which the method is also a unit of work.

6.5. Fast-read Annotations

While analyzing the performance impact on a benchmark that solves the traveling salesman problem (see Section 7), we noticed that the AJ version was significantly slower. Much of this slowdown was due to additional synchronization when reading a field that indicates the length of the best solution found so far. In the original Java version, this field was synchronized only for (relatively rare) updates. The original synchronization discipline was correct since the read of the field did not rely on its consistency with any other field. A similar situation would arise in any optimal solution search problem, and certainly in other contexts as well. Therefore, we generalized this pattern and updated our AJ implementation to allow programmers to indicate when certain field reads can be optimized. The typechecker enforces the following discipline:

```
class MinSolutionSoFar {
  atomicset(m);
  atomic(m) fastread int minLength = Integer.MAX_VALUE;
  atomic(m) int[] minPath = new int[MAX_PATH_SIZE];
  void updateMin(int newPathLength, int[] newPath) {
    if (newPathLength < minLength) {
      for (int i =0 ; i < newPathLength; i++) minPath[i]=newPath[i];
      minLength = newPathLength;
    }
  }
  int getMinSoFar(){ return minLength; }
}
```

generates the following code:

```
class MinSolutionSoFar {
  OrderedLock $lock_m;
  volatile int minLength = Integer.MAX_VALUE;
  int[] minPath = new int[MAX_PATH_SIZE];
  void updateMin(int newPathLength, int[] newPath) {
    synchronized($lock_m){
      if (newPathLength <  minLength) {
        for (int i =0 ; i <  newPathLength; i++) minPath[i]=newPath[i];
        minLength = newPathLength;
      }
    }
  }
  int getMinSoFar(){ return minLength; }
}
```

Fig. 20: Fast-read example.

(1) Any number of fields in an atomic set can be annotated as fastread.
(2) No unit of work may write to a fastread field more than once, nor may it write to more than one fastread field from the same atomic set.

The AJ code generator may then leave unsynchronized those units of work whose only access to an atomic set is a single read of a single fastread field. The code generator also marks all fastread fields as volatile in order to ensure that a thread that repeatedly reads a fastread field will eventually see updates from other threads. Figure 20 shows a slightly simplified version of a class from the AJ version of the traveling salesman problem along with the generated code.

The typechecker currently enforces condition 2 above using a simple, conservative, intra-procedural analysis. A straightforward effect system could be added to maintain modularity and make the analysis less conservative, but was not needed for our benchmarks. The fast-read optimization can be extremely beneficial: we observed a speedup of over 60× for the traveling salesman problem when compared to the AJ version without the minLength field annotated as fastread.

6.6. notunitfor Annotation

In order to preserve atomic-set serializability at run time, our type system needs to make conservative assumptions about which atomic sets an invoked method might access. Consider the example in Figure 21. Here, class Foo declares an atomic set F that protects a field count. Method f() first calls aStaticMethod() and then calls the

```
class Foo {
  atomicset(F)
  atomic(F) long count = 0;
  void f() { aStaticMethod(); g(); }
  void g(){ count++; }
  static void aStaticMethod(){ globalFoo.g(); }
  static Foo globalFoo;

  public static void main(String[] args){
    globalFoo = new Foo(); globalFoo.f();
  }
}
```

Fig. 21: Example for complex nesting of atomic set access

instance method g() on the current object. The question is now, whether f() needs to be synchronized. On the surface, it contains only a single method call that could access the atomic set F, so it should be sufficient to synchronize the call to g(). However, examining the implementation of aStaticMethod(), reveals that it accesses the current object's atomic set using an alias that was stored in a global variable. Thus, in order to ensure atomic-set serializability, the entire body of f() needs to be synchronized, such that it appears atomic to other threads that might, e.g., call g() concurrently.

While further analysis could detect such aliasing situations, we decided to make code generation conservative so as to always acquire the lock for the respective atomic set when a method calls another method. We found the overhead induced by this measure to be acceptable for the vast majority of cases. SPECjbb was the only benchmark that contained a complex situation where the over-synchronization introduced excessive slowdown (see the evaluation in Section 7). For cases where further analysis or manual inspection determines that introducing synchronization is unnecessary for atomic-set serializability, we introduce the notunitfor annotation. A method with that annotation will not acquire the lock(s) associated with atomic sets declared in its declaring class. As an example, all private methods of a class are implicitly annotated with notunitfor, as they can only be called from other methods in the same class that already synchronized on the appropriate atomic sets.

7. EMPIRICAL EVALUATION

To evaluate the AJ language design, we performed several experiments. First, we conducted an experiment in which we created AJ versions of a significant number of classes from the Java Collections Framework; Section 7.1 reports on the annotation overhead and effort involved. Second, we manually refactored several multi-threaded Java applications into AJ; Section 7.2 reports on the annotation overhead and issues encountered during these experiments. Finally, Section 7.3 reports on a number of performance measurements using an AJ version of SPECjbb, a well-known performance benchmark.

7.1. Java Collections Framework

As a first experiment, we investigate the effort involved in using atomic sets to create properly synchronized versions of representative classes from the Java Collections Framework. Specifically, we selected ArrayList, LinkedList, HashMap, LinkedHashMap, LinkedHashSet, HashSet, and TreeMap from package java.util in Sun's JDK 1.5 class libraries, along with any types on which these classes transitively depend. Each of these classes depends on several supertypes as well as several auxiliary classes (e.g., TreeMap declares nested classes SubMap and EntryIterator, as well as several anonymous nested classes). In total we included 63 types comprising 10,338 LOC. The collection classes we consider here do not contain any synchronization and are assumed to be used in conjection with synchronization wrappers[2] when accessed concurrently.

Determining the placement of atomicset and atomic annotations was straightforward. The collection classes we consider are comprised of 5 distinct inheritance sub-hierarchies, and we introduce one atomic set in each of the types Collection, Map, Iterator, LinkedList_Entry, and Map_Entry, which are the roots of these sub-hierarchies. All instance fields were added to the atomic set that we introduced for the sub-hierarchy in which its declaring class occurs. This is accomplished by adding an atomic annotation to the class declaration. We placed unitfor annotations on constructors that take other collections as an argument, on "bulk" methods such as addAll(), and on equals() methods in order to avoid concurrency bugs that could otherwise arise if the collection object that is passed as an argument is modified concurrently during the manipulation of the collection object pointed to by this. Such concurrency-related bugs are known to be problematic in the Java Collections Framework, as was previously pointed out by several researchers [Flanagan and Freund 2000a; Wang and Stoller 2006a; Hammer et al. 2008]. Our approach completely avoids them.

Introducing alias annotations required somewhat more thought, as this involves atomic sets in two classes. For example, the allocation of an AbstractList_ListItr object in class AbstractList was annotated as follows: new AbstractList_ListItr |I=this.L|(...), indicating that atomic set I in the newly created iterator-object is aliased with atomic set L in the list pointed to by this. Of the classes we annotated, only LinkedList_Entry was made internal. Map_Entry could not be made internal because it is exposed to client code via methods such as Map.entrySet() that provide a direct view on the map. Our type system prohibits this as internal types cannot be returned by public methods.[3]

The introduction of annotations required a few minor textual code changes. In particular, atomic fields must be accessed through accessor methods. Making LinkedList_Entry internal caused the LinkedList.addBefore() method to be rejected by our type-checker as it returned an internal class. This method could not be made private because it was invoked by LinkedList_ListItr.add(). However, as add() ignored the return value of this method call, we resolved the problem by creating a method addBefore2() with identical functionality as addBefore(), but with return type void.

On the whole, the amount of effort that was needed to create AJ versions of the collection classes was manageable. Ignoring the time that was spent to eliminate the Java features (generics and nested classes) that our implementation does not support yet, we estimate that it took us a few days to convert the 63 classes under consideration into AJ. Most of this time was spent on understanding the workings of the collection classes, and only a small fraction of the time was spent on inserting the new AJ language constructs. We conjecture that, for code developed from scratch, the amount of effort involved in writing AJ code is the same or less than that of writing properly synchronized Java code.

[2] Synchronization wrappers are objects that add synchronization to an existing collection. They define the same methods as the collection that they "wrap around". These methods synchronize on a lock object that is associated with the wrapper and then delegate the operation to the underlying collection. In Java, standard synchronization wrappers are provided in class java.util.Collections.

[3] One could clearly work around this issue by making Map.entrySet() return a map with copies of map entries and a link back to the original collection to handle mutations. However, this would have substantial cost. In general, the way the Collections API exposes mutable, derived data structures creates situations where multiple distinct-seeming data structures are in fact linked in complex way such that operations on one can result in failures in the other. Especially for concurrent code, this would ideally be avoided.

benchmark	LOC	files	sync	atomic sets	atomic (class)	atomic (field)	unitfor	alias (ref.)	alias (array)	not-unitfor	total
collections	10846	63	N/A	0	5	0	53	330	40	0	428
elevator	609	6	8	0	1	0	0	6	0	0	7
tsp	754	6	6	0	2	9	0	0	0	0	2
weblech	1971	14	8	2	0	4	0	0	0	0	6
jcurzez1	6639	49	58	5	2	7	15	23	1	0	53
jcurzez2	6633	49	48	4	3	2	6	3	1	0	19
cewolf	14002	129	14	0	6	0	0	2	0	0	8
SPECjbb (naive)	17639	64	187	0	18	0	2	13	24	0	57
SPECjbb (tuned)	17730	64	187	2	15	34	1	0	24	4	80

Table I: Annotations required to create AJ versions of several Java applications. The table shows, for each subject program, the number of lines of code, files and synchronized blocks that were present in the Java version. The subsequent columns count the number of annotations of each type, and the last column counts the total number of data-centric annotations.

The first row of Table I classifies the annotations in the 63 annotated classes. As mentioned, these classes did not contain any synchronization originally, hence the 'N/A' in the column that counts the number of synchronized blocks. As the table shows, we need a total of 428 annotations in 63 classes comprising 10,846 LOC. The majority of these annotations are related to ownership (aliasing), due to the pervasive use of iterators and auxiliary data structures such as list entries. This amounts to approximately 40 annotations per KLOC of source code, which is somewhat higher than the annotation overhead of the type systems by Flanagan et al. that guarantee race-freedom [Flanagan and Freund 2000a; Abadi et al. 2006] or atomicity [Flanagan and Qadeer 2003]. However, in our case, we *generate* properly synchronized code and guarantee serializability from these annotations alone, whereas Flanagan et al. require a program that is *already synchronized* using Java's synchronized construct.

7.2. Refactoring Java Applications into AJ

In order to validate our approach further, we manually refactored several multithreaded Java applications into AJ. The bottom 8 rows of Table I show some key characteristics of these applications, as well as the the number of data-centric annotations of each type that we needed to introduce. The *elevator* and *tsp* benchmarks have been used by several other researchers in projects related to data race detection (see e.g., von Praun and Gross [2004]). The *weblech* program[4] is a web crawler that recursively downloads all pages from a web site. The *jcurzez* program is a Java version of the popular *ncurses* program which allows building text-based user interfaces for simple terminals. Since the original *jcurzez* code did not have clearly defined support for multi-threading, we first created two new Java versions of the code with well-defined behavior in the presence of concurrency: *jcurzez1* achieves this behavior in a coarse-grained fashion while *jcurzez2* does so using more fine-grained synchronization. The *cewolf*[5] program is framework for creating various types of graphical charts. Finally, SPECjbb is a widely used multi-threaded performance benchmark[6].

The columns labeled "LOC", "files", and "sync" of Table I report the number of lines of source code, the number of files, and the number of synchronized blocks that were present in the Java versions of these programs. As can be seen from the number of data-centric annotations reported for the subject programs in Table I, the annota-

[4] See `http://weblech.sourceforge.net`.
[5] See `http://cewolf.sourceforge.net`.
[6] See `http://www.spec.org/jbb2005`.

tion overhead ranges between approximately 0.6 annotations per KLOC for *cewolf* to 11.5 annotations per KLOC for *elevator*. For the largest three applications, *jcurzez*, *cewolf*, SPECjbb (naive), and SPECjbb (tuned) the annotation overhead is a manageable 8.0/KLOC, 0.6/KLOC, 3.2/KLOC, and 4.5/KLOC, respectively. As can be seen from Table I, in all cases the number of data-centric annotations is less than the number of synchronized blocks that were present in the original Java versions. These results are highly encouraging because they suggest that data-centric synchronization combines reduced annotation overhead with a correctness guarantee that standard synchronized blocks do not offer. With the exception of SPECjbb, where we spent a significant amount of time on performance tuning as will be discussed later, the amount of effort involved in converting the subject programs into AJ was quite manageable, and usually required a small number of hours, with most of this time spent on understanding the existing concurrency in the programs.

We conclude this section with a few remarks on specific issues that we encountered while refactoring the subject programs from Java into AJ. In most cases, the transformations were very straightforward, and required only minor refactorings such as extracting code fragments into methods so that our unitfor annotations could indicate the desired units of work.

7.2.1. jcurzez. The two versions of the *jcurzez* benchmark demonstrate that AJ is capable of expressing synchronization at different granularities. It is interesting to note that converting the fine-grained Java version (*jcurzez2*) to AJ was more natural than converting the coarse-grained version (*jcurzez1*). This is reflected in its lower annotation overhead. The coarse-grained Java version was very close to the original source code, but with additional synchronized blocks included to enforce reasonable multithreaded behavior. The fine-grained Java version required more changes to the original code, mainly making method-local copies of some helper data structures that might be concurrently updated. But, the resulting Java code was much more natural to convert to AJ because most objects were responsible for their own synchronization rather than being aliased to containing objects. Simple tests show that the level of concurrency in the AJ versions were roughly equal to their Java counterparts and that the fine-grained versions indeed allowed much more concurrency than the coarse-grained versions.

7.2.2. elevator. The *elevator* benchmark is another example where AJ encourages a more encapsulated style of object-oriented programming. The original *elevator* code had a `Controls` object whose methods directly accessed data fields of a set of `Floor` objects stored in an array. Before accessing the fields of a particular `Floor`, it would synchronize on that object. We found the cleanest way to convert this code was to first move some code from the `Controls` class into `Floors`, which arguably led to cleaner Java code. Once this refactoring was complete, converting to AJ was straightforward.

7.2.3. tsp. After creating an initial AJ version of *tsp*, we noticed that this version was significantly slower than the original Java version. Much of this slowdown was due to additional synchronization when reading a field that indicates the length of the best solution found so far. In the Java version of *tsp*, this field was synchronized only for (relatively rare) updates. The original synchronization discipline was correct since the read of the field did not rely on its consistency with any other field. This issue can be resolved by placing a *fast-read* modifier on the method, as discussed in Section 6.5.

7.2.4. SPECjbb. The SPECjbb benchmark simulates a server-side application with classes representing entities like companies, customers, warehouses, and performing activities such as generating orders and making deliveries. Customers are represented by driver threads and database storage is simulated using the TreeMap binary tree class. SPECjbb uses synchronized statements and methods for ensuring mutual exclusion during order processing and wait()/notify() for coordinated ramp up and shut down of threads. We studied the existing synchronization in SPECjbb's source code in order to understand how atomic sets could be introduced. In the course of this analysis, we observed several issues:

Inconsistent synchronization. Synchronization appears to be somewhat haphazard. For instance, class Customer initializes shared fields in its constructor and in setUsingRandom(). Some of these fields have synchronized accessors, whereas others, like address, have unsynchronized accessors. Several methods (e.g., TreeMapDataStorage.deleteFirstEntities()) should logically be executed atomically, but there is no synchronization to enforce this.

Redundant synchronization. Many accessor methods in class Stock are synchronized even though the accessed fields are written only once, in a method called only by the constructor (e.g., Stock.getId()).

Use of wait/notify. The wait() and notify() methods are used to implement barriers that coordinate the threads of the multiple warehouses so that they ramp up, run, and shut down in a synchronized manner.

Ownership issues. Several data structures rely on collections from the Java Collections Framework to store data. For example, TreeMapDataStorage relies on a TreeMap to store its data. As mentioned, several methods of this class (e.g., deleteFirstEntities()) should logically be executed atomically but do not contain synchronization to achieve this.

Our approach was to add atomic sets in a straightforward way. Since we did not know the exact semantics of SPECjbb and the benchmark does not perform meaningful self-checking, we assumed that it was correct and verified that any synchronized section in the original code would be a unit of work in the AJ version. This check was done manually, by comparing the translated AJ code to the original benchmark. The atomic set annotations solved the issue of inconsistent synchronization mentioned above, as all accesses to fields that are part of an atomic set are guaranteed to be protected. For the ownership issue related to collections, our code reused the AJ versions of the collections of Section 7.1. Dealing with wait()/notify() required a bit of work as care is required to avoid deadlocks when calling wait(). We refactored SPECjbb to contain a dedicated barrier class that has a single atomic set and that uses the AJ wait()/notify() construct previously discussed in Section 6.1. Our compiler translates this construct to wait()/notify() calls on the generated lock object associated with this atomic set.

In the next section we will discuss further changes to the AJ version of SPECjbb that were required to obtain decent performance. For comparison, we will henceforth refer to the AJ version of SPECjbb discussed above as the *naive* AJ version of SPECjbb.

7.3. Performance Experiments

After writing the *naive* AJ version of the SPECjbb benchmark, we examined the overhead our naive conversion induced. We found that the AJ version scaled almost linearly up to at least 25 cores, with throughput ranging from 81.9% to 77.7% of the original version.[7] However, with more cores, the throughput of the naive AJ version degraded significantly, reaching only 13.8% of that of the original Java version at 98 threads.

Therefore, we investigated how the performance of the AJ version of SPECjbb could be improved, by examining the synchronization operations it performed at run time.

[7] All performance measurements reported in this section were taken on an Azul Vega 3 Series 3300 with two 54-core processors using 30GB of RAM with Azul's Java 1.6.0_07-2. On this machine, 10 cores are typically reserved for OS purposes, so our experiments are performed with up to 98 threads.

After some profiling, we identified SPECjbb's maps as a bottleneck and found that these were not synchronized in the original Java version. We investigated why this is the case and found that calls that access such a data structure are either already synchronized, or the data structure is read-only after initialization (which happens before threads are started). In such cases, the Java memory model guarantees that not having the data structure synchronized is safe. Therefore, we removed the atomic set from the map class altogether, and the read-only fields from the atomic sets of the classes that contained them. We will refer to the resulting AJ version of SPECjbb as the *basic* AJ version of SPECjbb.

This *basic* version of SPECjbb scaled up to about 30 threads with throughput ranging from 80.5% to 90.3% of that of the original Java version of SPECjbb. However, with more than 30 threads, the throughput of the basic AJ version degraded again, reaching only 28.0% of that of the original Java version at 98 threads.

Further profiling revealed that our AJ compiler still inserted synchronization operations in 4 methods that accessed only read-only maps, which were no longer declared in the atomic set of the respective class. This is due to the fact that the compiler must conservatively assume that such calls may access a field of the current atomic set (see the discussion in Section 6.6) As the memory model does not require synchronization to access read-only data, we annotated these 4 methods with notunitfor to obtain the *tuned* AJ version. This version scales well to 98 threads, as we will discuss shortly. Table I shows the annotation overhead for both the *naive* and the *tuned* versions SPECjbb, which is very similar. These results show that tuning a program with data-centric synchronization does not need to affect annotation overhead significantly.

Figure 22 compares the performance of the original Java implementation of SPECjbb to that of the *naive*, *basic* and *tuned* AJ implementations. It reports the number of SPECjbb2005 bops, which is a measure of the number of transactions per second, obtained from 2-minute runs with increasing numbers of threads (ranging from 1 to 98) for each version. From these measurements, it can be seen that, for a single thread, the *naive* AJ implementation of SPECjbb achieves a throughput of approximately 81.9% of that of the *original* implementation and that the *basic* AJ implementation of SPECjbb achieves a throughput of approximately 90.3% of that of the *original* implementation. The *tuned* implementation performs the same, reaching 90.2% of the throughput of the original implementation. The graph shows that the naive AJ version scales up to about 30 threads, but degrades significantly with more than 40 threads, while the basic AJ version scales only up to about 30 threads and degrades only slightly from there. Specifically, for the situation with 98 threads, we measure a throughput of 13.8%, 28.0%, and 90.8% of that of the *original* Java version, for the *naive*, *basic* and *tuned* versions, respectively. The remaining overhead of the *tuned* version can be attributed to some of the additional locking introduced by atomic sets, which at the same time renders the synchronization much more consistent and thus safe.

8. CONCLUSIONS

We have presented a type-based approach for data-centric synchronization, based on atomic sets and units of work. Our new type system guarantees atomic-set serializability while enabling separate compilation and atomic sets that span multiple objects. We implemented this approach in AJ, a significant subset of Java extended with atomic sets, and created an AJ-to-Java compiler. We demonstrated that our approach has low annotation overhead, by manually rewriting into AJ several classes from the Java Collections Framework, and a set of Java applications that includes SPECjbb, a widely used multi-threaded performance benchmark.

In our experiments, the annotation overhead was approximately 40 annotations for each KLOC of source code in Java Collections, and ranged from 0.6 to 11.5 annotations

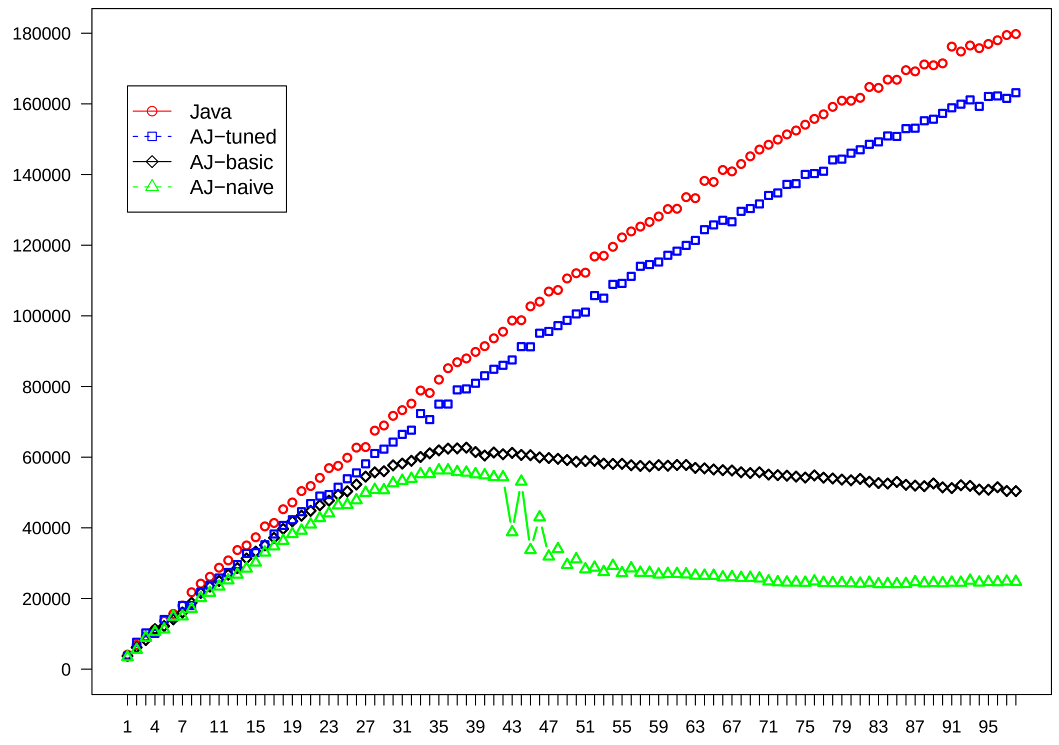

Fig. 22: Performance measurements for SPECjbb. The figure shows the number of bops (a measure of throughput, higher is better) achieved by the *original* Java code, and by the *naive*, *basic*, and *tuned* AJ versions, for up to 98 threads.

per KLOC for the other applications. In each of these applications, this amounted to fewer annotations than the number of synchronized blocks that were present in the original Java version. Our performance experiments with SPECjbb revealed that the *naive* AJ version did not perform well. However, with some minor performance tuning we were able to achieve nearly the same performance as the original Java version.

We expect SPECjbb to be representative of the majority of user written code where concurrency concerns are only a small part of the code. As performance optimizations were not the main focus of this work we consider the reported results to be an encouraging indication that our approach is capable of generating code with acceptable performance while providing a correctness guarantee that Java's current synchronization mechanism does not offer.

In future work, we plan to explore several avenues for improving performance, including the use of program analysis to tighten the scope of synchronization. We also plan to explore the use of static analysis for detecting possible deadlock.

Additional information about this project at http://sss.cs.purdue.edu/projects/aj.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Grants CCF 1048398, CCF 0938232, CNS 0716659 and CCF 1048398. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

ABADI, M., FLANAGAN, C., AND FREUND, S. N. 2006. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems 28,* 2.

ARTHO, C., HAVELUND, K., AND BIERE, A. 2003. High-level data races. *Softw. Test., Verif. Reliab. 13,* 4, 207–227.

BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. 2010. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 53–64.

BOCCHINO, R., ADVE, V., DIG, D., ADVE, S., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. 2009. A type and effect system for deterministic parallel Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 97–116.

BOYAPATI, C., LEE, R., AND RINARD, M. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

BOYAPATI, C. AND RINARD, M. 2001. A parameterized type system for race-free Java programs. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

BURROWS, M. AND LEINO, K. R. M. 2004. Finding stale-value errors in concurrent programs. *Concurrency - Practice and Experience 16,* 12, 1161–1172.

CEZE, L., VON PRAUN, C., CASCAVAL, C., MONTESINOS, P., AND TORRELLAS, J. 2008. Concurrency control with data coloring. In *Workshop on Memory Systems Performance and Correctness (MSPC)*. 6–10.

CHEREM, S., CHILIMBI, T., AND GULWANI, S. 2008. Inferring locks for atomic sections. In *Conference on Programming Language Design and Implementation (PLDI)*.

CLARKE, D., POTTER, J., AND NOBLE, J. 1998. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming, Languages, and Applications (OOPSLA)*.

DENG, X., DWYER, M. B., HATCLIFF, J., AND MIZUNO, M. 2002. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering (ICSE)*. 442–452.

ENGLER, D. R. AND ASHCRAFT, K. 2003. Racerx: effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles (SOSP)*. 237–252.

FLANAGAN, C. AND FREUND, S. 2000a. Type-based race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*.

FLANAGAN, C. AND FREUND, S. N. 2000b. Type-based race detection for Java. In Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). 219–232.

FLANAGAN, C., FREUND, S. N., LIFSHIN, M., AND QADEER, S. 2008. Types for atomicity: Static checking and inference for Java. *ACM Transactions on Programming Languages and Systems 30,* 4.

FLANAGAN, C. AND QADEER, S. 2003. A type and effect system for atomicity. In *Conference on Programming Language Design and Implementation (PLDI)*.

GREENHOUSE, A. AND BOYLAND, J. 1999. An object-oriented effect system. In *European Conference on Object Oriented Programming (ECOOP)*.

GROTHOFF, C., PALSBERG, J., AND VITEK, J. 2007. Encapsulating objects with confined types. *Transactions on Programming Languages and Systems 29,* 6, 32–73.

HAMMER, C., DOLBY, J., VAZIRI, M., AND TIP, F. 2008. Dynamic detection of atomic-set-serializability violations. In *ICSE*.

HARRIS, T. AND FRASER, K. 2003. Language support for lightweight transactions. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 388–402.

HERLIHY, M. AND MOSS, J. E. B. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture (ISCA)*.

HOARE, C. A. R. 1974. Monitors: an operating system structuring concept. *Communications of the ACM 17,* 10, 549–557.

KIDD, N., REPS, T., DOLBY, J., AND VAZIRI, M. 2011. Finding concurrency-related bugs using random isolation. *International Journal on Software Tools for Technology Transfer (STTT) 13*, 495–518.

KULKARNI, A., LIU, Y. D., AND SMITH, S. F. 2010. Task types for pervasive atomicity. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 671–690.

LAI, Z., CHEUNG, S. C., AND CHAN, W. K. 2010. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering (ICSE)*. 235–244.

LEINO, K. R. M. 1998. Data Groups: Specifying the modification of extended state. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

LEINO, K. R. M., SAXE, J. B., AND STATA, R. 1999. Checking Java programs via guarded commands. In *ECOOP Workshop Reader*. 110–111.

LU, S., PARK, S., HU, C., MA, X., JIANG, W., LI, Z., POPA, R. A., AND ZHOU, Y. 2007. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Symposium on Operating Systems Principles (SOSP)*. 103–116.

LU, S., PARK, S., SEO, E., AND ZHOU, Y. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 329–339.

LUCIA, B., CEZE, L., AND STRAUSS, K. 2010. Colorsafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *International Symposium on Computer Architecture (ISCA)*. 222–233.

MCCLOSKEY, B., ZHOU, F., GAY, D., AND BREWER, E. 2006. Autolocker: Synchronization inference for atomic sections. In *Symposium on the Principles of Programming Languages (POPL)*.

NOBLE, J., POTTER, J., AND VITEK, J. 1998. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)*.

O'CALLAHAN, R. AND CHOI, J.-D. 2003. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 167–178.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. E. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Symposium on Operating Systems Principles (SOSP)*. 27–37.

VAZIRI, M., TIP, F., AND DOLBY, J. 2006. Associating synchronization constraints with data in an object-oriented language. In *Symposium on the Principles of Programming Languages (POPL)*.

VITEK, J. AND BOKOWSKI, B. 2001. Confined types in Java. *Software Practice & Experience 31,* 6, 507–532.

VON PRAUN, C. AND GROSS, T. R. 2004. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology 3,* 6, 103–122.

WANG, L. AND STOLLER, S. D. 2006a. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Principles and Practice of Parallel Programming (PPoPP)*.

WANG, L. AND STOLLER, S. D. 2006b. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering 32,* 2.

WRIGSTAD, T., PIZLO, F., MEAWAD, F., ZHAO, L., AND VITEK, J. 2009. Loci: Simple thread-locality for Java. In *European Conference on Object Oriented Programming (ECOOP)*.

XU, M., BODIK, R., AND HILL, M. D. 2005. A serializability violation detector for shared-memory server programs. In *Conference on Programming Language Design and Implementation (PLDI)*. 1–14.