The Non-Linearity of `volatile` in Java™

John Boyland*
University of Wisconsin-Milwaukee, USA
boyland@cs.uwm.edu

ABSTRACT

Linear logic and related logics (such as separation logic and fractional permissions) have proven useful in verifying concurrent programs because they make it easy to reason about heap separation properties. However "volatile" fields in Java are difficult to reason about in strictly linear fashion. Volatile is much more easier handled using non-linear concepts such as immutability and ownership.

1 INTRODUCTION

Concurrent programs in which threads share mutable state use synchronization and/or volatile fields to ensure thread safety. In Fig. 1, a map implementation synchronizes modifications to the map (as in `put`), but does not synchronize read-only access (as in `get`). Avoiding synchronization in the common, non-modifying case, can improve concurrency. This is thread-safe only because (1) access to the underlying map is through a `volatile` field, and (2) the underlying map is left immutable. Immutability means that the underlying map is never mutated while it is being read. Volatility ensures that `get` always gets the latest version of the map.

Figure 2 gives another example of using `volatile`: here multiple threads may generate data (using `push`) and a single thread periodically polls this data (using `pull`). The sample thread may miss data that arrive too quickly, but does not receive multiple copies of a single event. Even though it never uses synchronization, this code is thread-safe under the assumption that only one thread samples the data. In the case of multiple sample threads, the update of the `collected` flag can lead to a data race.

This paper informally discuss how such uses of volatile fields can be shown thread-safe through program analysis. Section 2 discusses the difficulties faced when using concurrent separation logic. Then Sect. 3 describes how "ownership" type systems solve the problem.

2 LINEARITY

Concurrent Separation Logic (CSL) [4] uses a variant of linear logic to reason about concurrent programs. Programs can be shown thread-safe using local reasoning alone by separating the heap among threads. The separation is not fixed, and threads can transfer access to state through mutual exclusion blocks: `l.acq(); ...; l.rel()` or more simply `atomic { ... }`. After acquiring the lock or entering the atomic block, the thread gets access to a portion of the heap indicated by a "shared invariant." The invariant must be re-established at the point the lock is released or the atomic block is exited, and the access of this portion of the heap is removed from the thread. Since the invariant may include

```
public class RarelyChangingMap<K,V> {
  volatile Map<K,V> immMap;
  ⋮
  public V get(Object k) {
    return immMap.get(k);
  }

  public V put(K key, V value) {
    synchronized (this) {
      Map<K,V> copy = new HashMap<K,V>(immMap);
      V old = copy.put(key,value);
      immMap = copy;
      return old;
    }
  }
}
```

Figure 1: Using `volatile` for a thread safe `Map`.

```
public class Sampler {
  public static class Datum {
    public int x, y;
    boolean collected;
    public Datum(int x, int y) {
      this.x = x; this.y = y;
    }
  }
  private volatile Datum datum;

  public void push(int x, int y) {
    datum = new Datum(x,y);
  }

  public Datum pull() {
    Datum d = datum;
    if (d.collected) return null;
    d.collected = true;
    return d;
  }
}
```

Figure 2: Sampling data from multiple threads.

*Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

existentials, it may refer to a different section of the heap than it did before.

Extending this model to volatile variables can be done by converting volatile variable accesses into atomic blocks, and then adding extra actions inside the atomic blocks. For example the code in Fig. 1 can handled by using existentially quantified fractions [2] to model immutability. Inside the atomic block, the existential is opened, the fraction is divided in half, and then it is re-packaged.

Figure 2 can be handled if the atomic block is extended to include the assignment to `d.collected`. Alternately, auxiliary state [9] can be used: a new "model" field is added to the object that has value NEW, READ or OLD. When in the NEW state, the field comes with permission to write the `collected` field. This permission is removed if the field is read in this state, in which case the state is changed to READ. Then if the state still READ at the time the `collected` field is set, it is changed to OLD and is accompanied with an immutable permission for `d.collected`. If the field is read in the OLD state, a fraction is split off to be used in the conditional return. Moving the state to OLD requires a new atomic block just for updating the auxiliary state. Since the model field is added just for verification but not present at runtime, the empty atomic block can then be optimized away.

This use of atomic blocks to model `volatile` can be justified because at the hardware level, volatile accesses cannot be simultaneous. However, it is unsatisfying. If one desires a more high-level description that avoids auxiliary state, it seems necessary to use nonlinear reasoning. The field could be read not at all between writes, or once or multiple times. The ability to use some fact not at all, once or multiple times is a characteristic feature of traditional logic as opposed to linear logic.

Separation logic can be defined on top of the logic of bunched implications [8] which includes both linear and nonlinear aspects. Indeed even Girard's original linear logic [7] has the `!` and `?` modalities which permit a logical term to be duplicated or ignored. The non-linear predicates available however in CSL do not refer to the heap.

3 OWNERSHIP

In an ownership type system (such as SafeJava [1]), each object is assigned an owner which never changes. A concurrent program is thread-safe if all accesses to (potentially) mutable fields occur dynamically inside of a synchronization on the object whose field is accessed or any of its (transitive) owners. Thread safety depends on encapsulation so that no object is ever accessed outside of the dynamic context of its owner (owners as dominators, as formalized by Clarke [5]). Volatile fields can be handled as a loop hole—volatile field accesses are unchecked. This is thread-safe because any access of the state reached through the volatile field must be in the context of its owner.

In principle, the second example (Fig. 2) can be handled by indicating the objects of type `Datum` belong to the reading thread once they are created.

The first example is also amenable to analysis using an ownership system provided that the system includes a concept of immutability (e.g. immutable objects have a distinguished owner).

The relative ease with which these examples can be handled (`volatile` simply is an escape hatch) is due to the permanent (non-linear) nature of ownership in these systems. Ownership transfer in defined in some systems, but always involves a linear (and hence flow-sensitive) aspect which brings back the complexity of linear systems. Sometimes indeed the complexity exceeds that of CSL.

We have been investigating how to combine nonlinearity (ownership) and linearity (uniqueness) at a formal level by building on adoption [6] which formalizes ownership. In adoption, a mutable object is irrevocably adopted by another object. Then whenever one has access to the "parent," one also has (indirect) access to the "child." Direct access requires that the child be "focused" upon. Adoption is a *global changing* non-linear property of a program, although change is only in one direction. New adoption relations can be added, they never go away. We have defined a kernel logic that supports linear-style permissions, fractions and adoption (generalized as "nesting") [3].

4 CONCLUSIONS

I first introduced two examples of `volatile` usage and explained the difficulties in using a linear system to verify them. Then I explained how they can be handled more easily in an ownership type system. I hope therefore to increase interest in how one can formalize ownership as a nonlinear subsystem in a mostly linear logic.

REFERENCES

[1] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Ph.D., MIT, February 2004.

[2] John Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SAS '03*, volume 2694 of *LNCS*, pages 55–72. 2003.

[3] John Boyland. Semantics of fractional permissions with nesting. Technical Report CS-07-01, University of Wisconsin–Milwaukee, Department of EE & CS, December 2007.

[4] Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1–3):227–270, 2007.

[5] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, 2001.

[6] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI '02*, pages 13–24. May 2002.

[7] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50(1):1–102, 1987.

[8] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL '01*, pages 14–26. 2001.

[9] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. In *POPL '07*. 2007.