IT 11 024
Examensarbete 30 hp
May 2011

The Design, Implementation and Evaluation of a Pluggable Type Checker for Thread-Locality in Java

Amanj Sherwany

Institutionen för informationsteknologi
Department of Information Technology

To mom, dad
& my beloved girlfriend

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

The Design, Implementation and Evaluation of a Pluggable Type Checker for Thread-Locality in Java

Amanj Sherwany

This thesis presents a simple type system for expressing thread-locality in Java. Classes and types are annotated to express thread-locality and violations, where supposedly thread-local data may be shared between two or more threads, are detected at compile-time. The proposed system is an improvement over Loci, a minimal and modular type checker for expressing thread-locality in Java due to Wrigstad et al.
The improved Loci system presented in this thesis only adds an additional metadata annotation, four in total. We implemented the system as a command line tool that can be plugged into the standard javac compiler and used it to evaluate our design on a number of benchmarks.
We believe that Loci is compatible with how Java programs are written and that the improved system keeps the annotation overhead light while making it even simpler to treat a value as thread-local.

Handledare: Tobias Wrigstad
Ämnesgranskare: Konstantinos Sagonas
Examinator: Anders Jansson
IT 11 024
Tryckt av: Reprocentralen ITC

Acknowledgements

I thank my thesis adviser, Tobias Wrigstad, his continuous help, corrections and suggestions made my job a lot easier. Many thanks to my reviewer, Kostis Sagonas, for revising my work and his great suggestions.

I also thank the developers behind the Checker framework who work hard to develop such an awesome tool, and they had a very great online support. And Adam Warski who developed the Maven2 plugin for the Checker framework was also helpful, as I forked his work to develop a Maven2 plugin for Loci which is based on the Checker framework's Maven2 plugin.

My lovely girlfriend, Nosheen Zaza, was a great support during my work on this thesis, we used to have discussions about Loci together and most of the time I was considering her suggestions and directions, I really thank her. Also Sobhan Badeyozaman and Karwan Jaksi, who were my colleagues had supported me mentally. At the end I want to thank my sisters and brothers for their continuous support, thank you all.

Contents

List of Figures

List of Tables

Listings

CHAPTER 1
INTRODUCTION

> I believe we can have our cake and eat (most of) it too.
>
> Gilad Bracha, Pluggable Type Systems

1.1 Background

Manually verifying properties of programs is hard and prone to errors. In this light, programmers need tools that help them express their intentions in a way that can be checked automatically, preferably at compile-time. Thread-locality is an often desirable property of data objects in parallel or concurrent programs. Knowing that some of the objects that are manipulated by a system are thread-local relieves programmers from worrying about concurrency control for those parts of the system [21]. In this thesis, we report on the design, implementation and evaluation of a programmer tool for expressing and statically checking thread-locality in Java programs to facilitate correct parallel and concurrent programming.

1.1.1 Thread-Locality

Today, most computers have more than a single core and are powerful enough to run several tasks concurrently. To make use of this power, programs should be designed with parallelism in mind and most of today's programming languages have built in support for multi-threading or other forms of parallelism [20].

Java has a built-in support for multi-threading, which requires library developers to consider issues of thread-safety in their library designs, which

complicates programming and at worst break compositionality of code due to conflicting locking behaviour.

Thread-local values make parallel and concurrent programming easier, since accesses to thread-local data are sequential and thereby relatively easy to reason about. For example, there will never be data races or dead locks on thread-local data. There are several other advantages of thread-locality: in real-time systems, thread-locality avoids lock inflation which is important when calculating worst-case run-times/paths, garbage collectors for thread-local data can execute in parallel with the rest of the system, and, finally, manipulation of thread-local data does not need synchronization with main memory since there can be no other witnesses to a change [21].

Most mainstream programming languages lack support for expressing thread-locality and programmers are forced to fall-back to documentation or unverified source-level comments. Conservatively, we must consider any accesses to a field in the program as potentially violating intended thread-locality. The only way to "guarantee" that supposedly thread-local data stays thread-local (in our terminology, does not leak or escape the designated thread) is by ocular inspection of the source code of potentially many classes in a program.

Java has supported thread-local fields through its `ThreadLocal` API [7] since version 1.2 [15]. These utility classes allow defining fields for which each accessing thread has its own copy and consequently access the field in a race-free manner. However, nothing prevents the contents of the field to be shared across threads. In this respect, the "protection" that is offered by using the `ThreadLocal` class is similar to that of name-based encapsulation [21].

A Java compiler does not enforce proper use of library code from a semantic perspective. Where thread-locality is concerned, Java programs do not prevent reading thread-local fields and storing the results in globally accessible locations, which might violate thread-locality altogether. The only way to obtain automatic guarantees for thread-locality is by using *only* thread-local fields which is cumbersome, prone to errors, and has an unreasonable overhead in terms of field size and field access time. In short, Java offers no support for avoiding thread-locality violations. In a program where thread-locality is important, programmers need tool support to express and check thread-locality.

1.1.2 Pluggable Type Checkers

Pluggable type checkers were proposed by Bracha [9] to allow static checking of different program properties at different stages of program development. Checks for different properties can be plugged in or removed or combined freely and, according to Bracha, have most of the advantages of mandatory type systems without most of the drawbacks [9]. In Bracha's terms, a pluggable type checker:

1. has no effect on the run-time semantics of the programming language, and

2. does not mandate type annotations in the syntax.

There are many frameworks for extending programming languages by custom pluggable type checkers [14, 11, 18]. Java 6 and Java 7 [5] both have basic support for pluggable type checkers, through support for metadata annotations that can be checked by the Annotation Processing Tool (`apt`) [1]. Starting with Java 8, a more expressive annotation system will subsume the existing one, along with a framework for the easy specification of pluggable checkers [6, 13].

There is an extensive development for Java's upcoming type checker [14, 11] called the Checker framework and a preview version of the system already exists.

1.2 Purpose and Goal

In this thesis, we extend earlier work by Wrigstad et al. [21] on Loci, a pluggable type checker for thread-locality in Java. Loci was designed with the following design goals in mind:

DG1: Conservative checking (soundness) It should detect all leaks that occur or might occur. False positives might ensue.

DG2: Backwards compatibility It should be possible and feasible to "port" legacy Java code to use Loci without requiring extensive rewrites.

DG3: Optional checking Makes applying the checker only on a part of the code possible.

DG4: Low annotation overhead Aimed to have a good selection for default annotations so the programmers need only a few number of annotations to express their intentions.

DG5: Fully defined The checker should cover all features of the targeted programming language.

However, the initial Loci system proposed by Wrigstad et al. took a number of shortcuts which compromised the design integrity of the system. This lead to many lost opportunities to express actual thread-locality, and several important Java idioms that could not readily be expressed.

This thesis improves the design of Loci, and evaluates its improvements through practical evaluations on well-known multi-threaded Java benchmark sources. Specifically, we make the following contributions:

C1: Support for generics We extended the design of Loci to support annotations on type parameters in generic classes. More details can be found in Sections 2.3.5 and 3.2.

C2: Locality-polymorphic methods We extended the design further to support method type parameters too. More details can be found in Sections 2.3.3 and 3.4.

C3: Changing default annotations We changed some of the default annotations, like: classes are "flexible" by default instead of `@Shared`. More in Section 3.3.

C4: Typing cloning and equals We annotated the **clone** and **equals** methods in **Object** class which is the parent of every Java class, thus constrains the interface of every occurrence of those two methods in every Java classes. More in Section 3.5.

C5: Implementation of a stand-alone checker We implemented Loci as a compiler plugin, using the Checker framework [11, 14] which reports a warning or an error as soon as it detects a thread-locality leak. More in Chapter 4.

C6: Design Evaluation We evaluated Loci by applying it on several standard benchmarks [8, 3]. More in Chapter 5.

We believe that the resulting system is compatible with how Java programs are written, and requires a low annotation overhead ($\sim$ 15 annotations/K-LOC), and improves on the previous Loci design by allowing unannotated classes to create thread-local values.

1.3 Outline

This thesis consists of six chapters. The second chapter introduces Loci and its core design concepts. The third chapter is a continuation of the second chapter, covering our improvements from Loci 1.0 to Loci 2.0.

The fourth chapter discusses the Loci tool and its dependencies and known bugs. Chapter 5, discusses the evaluation of our design using a set of well-known benchmarks. The last chapter concludes and outlines future work.

CHAPTER 2

INTRODUCING LOCI

> Make everything as simple as possible, but not simpler
>
> Albert Einstein

Having stressed the importance of thread-locality and the reasons for having a tool support to enable programmers to statically check their programs against thread-locality violations, we now present original Loci system that the thesis extends. The chapter starts by presenting Loci's logical view of memory, its annotations, and its core design. Finally the chapter ends with an example of applying Loci to real-world code.

2.1 Loci's Logical View of Memory

Loci *logically* divides the heap into a number "heaplets". Each thread has its own heaplet, and each heaplet has only one thread. There is also a heap which is shared among all the threads. A similar approach has been proposed also by Wrigstad et. al [21], Domani et. al [12] and Steensgaard et. al [19]. The Loci annotation system enforces the following simple properties on Java programs, shown in Figure 2.1[1]:

1. References from one heaplet into another are not allowed ($\longrightarrow$).
2. References from heaplets to the shared heap are unrestricted ($\dashrightarrow$).
3. References from the shared-heap into a heaplet must be stored in a thread-local field ($\bullet\!\!\dashrightarrow$).

[1] The figure is taken from Wrigstad et al. [21].

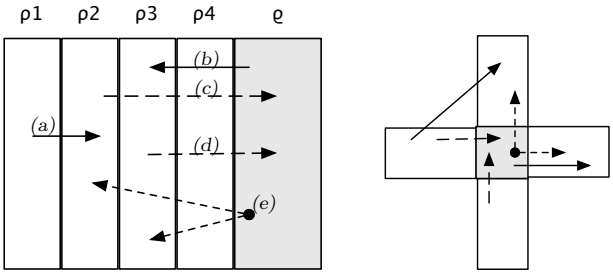

Figure 2.1: Thread-Local Heaplets and a Shared Heap. The teal area (ϱ) is the shared heap, white areas ($\rho_1..\rho_4$) represent the thread-local heaplets. Solid arrows are invalid and correspond to Property 1 in 2.1, dashed arrows are valid pointers into the shared heap (Property 2), respectively from the shared heap into heaplets (Property 3, when "anchored" in a bullet). The right-most figure is a Venn diagram-esque depiction of the same program to illustrate the semantics of the shared heap

The third property above ensures that a thread-local data (ρ_i) is only accessible by the thread i to which ρ_i belongs. But if j^{th} thread wants to access the same data, it will either get a reference to ρ_j (which belongs to j^{th} thread) or `null`. This property ensures that each active thread in the system has its own copy of each thread-local field. Therefore, it ensures that different threads access different thread-local fields [21].

Together, these simple properties make heaplets effectively thread-local, and objects in the heaplets are thus safe from race conditions and data races [21].

2.2 Annotations

Loci extends Java with three annotations[2]:

`@Local` which denotes a thread-local value;

`@Shared` which denotes a value that can be arbitrarily shared between threads; and

`@ThreadSafe` which denotes a value that must be treated in such a way that thread-locality is preserved, but the value may not be thread-local in practice. It is used to allow a limited form of parametricity and denotes an object whose thread-locality is not known and must therefore be treated conservatively.

Clearly, at run-time, every value is either thread-local or not. The purpose of `@ThreadSafe` annotation is to enable flexibility and variability in a

[2]In fact there are other annotations too, but these three are the core annotations in Loci.

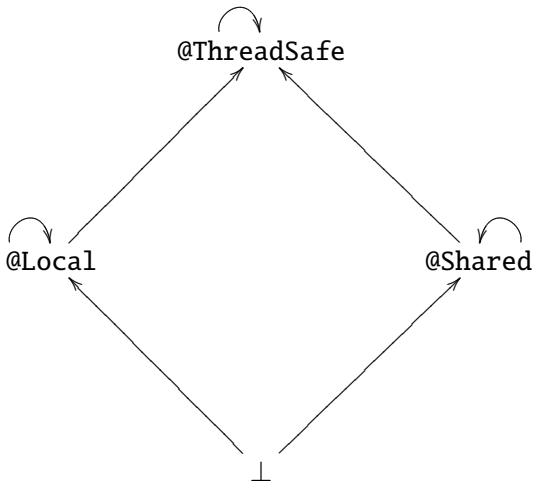

Figure 2.2: Annotation lattice capturing dataflow constraints. Each wedge denotes permitted dataflow. ⊥ denotes "free" objects.

design, e.g. methods that accept both `@Local` and `@Shared` values, such as Java's equals method, see Section 3.5.

Loci requires that dataflow preserves the thread-locality of a value in a variable, except for assignments into `@ThreadSafe`. This dictates what must hold if the contents of a variable y is stored into a variable, field or parameter x (by assignment, argument passing, etc.). For clarity, a summary is found in Figure 2.2.

Unlike `@Shared` and `@Local`, `@ThreadSafe` annotation cannot appear on class declarations and object instantiations. However, they all can appear on variable declarations, arrays, parameters, generics and method type parameters.

2.3 Integration with Core Java Concepts

2.3.1 Classes and Types

Classes in Loci can be `@Local` or `@Shared`. `@Shared` classes can only be instantiated as shared objects and `@Local` classes can only be instantiated as thread-local objects. A class is `@Local` (or `@Shared`) if the class, its super class or one of its implemented interfaces is annotated `@Local` (or `@Shared`). A class which has no explicit annotation and has no direct or indirect `@Shared` or `@Local` superclass (or implemented interfaces) is treated specially and can be used to create both shared and thread-local instances. These classes are useful for libraries and other situations where flexibility is desirable, in this thesis we call them "flexible classes".

`@Local` and `@Shared` classes may only be subclassed by `@Local` and `@Shared` classes respectively, while flexible classes can be subclassed by `@Local`, `@Shared` or flexible classes. In Loci, the `Object` root class is flexible, which is an important design choice to allow maximum flexibility, since it is the parent class of every valid Java class.

If `c` is a field, variable, return type or parameter and is declared without having an explicit annotation and is an instance of class `T` when `T` is a flexible class, then `c`'s thread-locality will be the same as the current instance/object that holds it, we call these `@Owner` types.

```
class Foo { //A flexible class
  Object f;
}

@Local Foo a = ...; //A thread-local instance of Foo
@Shared Foo b = ...; //A shared instance of Foo

//a.f is thread-local, and b.f is shared
```

2.3.2 Fields and Thread-Locality

`@Local` or `@ThreadSafe` fields are not allowed inside `@Shared` and flexible classes. This is necessary since the enclosing object might be shared across threads making the field effectively shared too. One way to have them is by implementing them through a `java.lang.ThreadLocal` indirection, which degrades performance.

```
@Shared class Foo{
    @Local Object a; //Invalid
    ThreadLocal<@Local Object> b; //OK
}
```

2.3.3 Statics and Thread-Locality

Class objects are shared by default (in Loci's eyes, completely ignorant of class loaders, etc.). In Java, the enclosing object of a static context is a class object. Therefore, every unannotated type with a flexible in static context defaults to `@Shared` (for reasoning refer to Section 2.3.1).

```
@Local class Foo {
  static Object f; //implicitly @Shared

  static {
    @Shared Object x = new Object();
    f = x; // OK, f is implicitly @Shared
  }
```

```
    static Object id(Object o1) { return o1; }
    static Foo fooId(Foo o2) { return o2; }
}
```

Listing 2.1: An example to show static methods, blocks and fields.

The above listing shows that the default annotations within a static context is `@Shared`. Considering the `id(Object o1)` method which takes an `Object` and returns an `Object`. We mentioned in Section 2.3.1, `Object` is a flexible class, therefore the thread-locality of its unannotated instances depend on their enclosing objects. But since Loci can trivially decide that the enclosing object is `@Shared`[3] the implicit annotations will be `@Shared`, which means the `id` method is identical to:

```
static @Shared Object id(@Shared Object o1) { return o1; }
```

In the above method, Java (as well as Loci) loses type information. If we want to keep the type information we should refactor the method to use polymorphic type argument:

```
static <T extends Object> T id(T o1) { return o1; }
```

One of our major contribution in this work was extending Loci with another annotation[4], called `@X`, which is used only within method type argument declarations. Upon method call, the `@X` annotations are replaced with the passed parameter's annotation[5]. There are several benefits of having this annotation, most importantly supporting for a method that can deal with both `@Shared` and `@Local` variables without losing the type information.

```
static <@X T extends Object> T id(T o1) { return o1; }
...

@Local Object b = Foo.id(new @Local Object()); //OK
@Shared Object b = Foo.id(new @Shared Object()); //OK
```

2.3.4 `@Owner` and Subtyping

As we mentioned in Section 2.3.1 an `@Owner` inherits its thread-locality from the object that contains it [22]. Therefore, an `@Owner` inside a `@Shared` object is `@Shared`, and an `@Owner` inside a `@Local` object is `@Local`.

```
class Foo{
  Object value; //Implicitly @Owner

  //The parameter type is implicitly @Owner
  void setValue(Object value){
```

[3] As they are within a static context.
[4] Listed as C2 in Section 1.2
[5] For more information refer to Section 3.4

```
    this.value = value;
  }

  //The return type is implicitly @Owner
  Object getValue(){
    return value;
  }
}

...
@Shared Foo aShared = new Foo();
@Local Foo bLocal = new Foo();

aShared.value = new @Local Object(); //Not OK, value is @Shared
bLocal.value = new @Shared Object(); //Not OK, value is @Local
```

Loci cannot guarantee the "actual" thread-locality of any `@ThreadSafe` type[6]. Therefore, Loci cannot determine the inherited thread-locality of any occurrence of `@Owner` types which are enclosed by a `@ThreadSafe` object. Imagine that we treat `@ThreadSafe` as `@Shared` and `@Local`, which means that all `@Owner` variables enclosed by a `@ThreadSafe` instance are `@ThreadSafe`, this makes the following program legal, despite leading to a serious leak.

```
class B{
  Object value; //value is implicitly @Owner
}

@Shared B b = ...; //b.value should be @Shared
@ThreadSafe B c = b; //if we suppose that c.value is @ThreadSafe
c.value = new @Local Object(); //Leak!
```

To solve this, we follow the same path as wildcards in Java [4]. A wildcard in generics is a readable but not writable property:

```
class Cell{
  Object value;
  void set(Object t){...}
  Object read(){...}
}
...
@Shared Cell b = ...;
@ThreadSafe Cell c=b;//Java(as well as Loci) loses type information
@ThreadSafe Object b = c.read(); //OK

b.value = new @Shared Object(); //OK
c.value = new @Local Object(); //value is not writable
c.set(new Object());//Is not OK
```

[6] At run time, everything is either `@Shared` or `@Local`.

A similar solution was suggested by Lu et al. [17] for their "Dynamic Exposure", for preventing exposing of an object by downgrading its dynamic ownership [17].

2.3.5 Arrays and Generics

Loci requires arrays to have the same thread-locality as their elements. Therefore, you can never have an array of mixed thread-locality[7]. The reason why mixed thread-locality is *not* supported is purely technical and slightly subtle. Assume `@Shared class A` and that we allow `@Local A[] f`. Now store `f` in a `@Local Object` and then downcast it to a `@Local Object[] f2`, all legal operations in a system that mimics Java. Now, there is no way of telling whether the elements of `f2` are shared or thread-local, since there is no thread-locality information for objects at run-time. An alternative design would be to use special annotations for arrays but this departs from standard Java subtyping—`@Shared Object` would not be the supertype of all shared objects, etc.—which would break many patterns in existing code.

For the system to remain simple and minimal, we chose to ban the creation of arrays with different thread-locality than their elements. A simple proxy object that wraps either the array or each element in the array can easily solve the problem.

Extending Loci to fully support generics was one of our most important contributions, listed as C1 in Section 1.2. In generics annotations are bounded similar to how types are bounded by the extends clause.

```
class A<@ThreadSafe T> {}
class B<@Shared K> {}
```

The type parameter `T` in class `A` is bounded by a `@ThreadSafe Object`, which means we can pass any thread-locality. However, class `B` only accepts `@Shared` instances as its type argument.

If a bound on annotations is not explicitly defined, the annotation on the (possibly implicit) upper class bound is used.

```
class A<T extends @ThreadSafe Object> {}
class B<K extends @Shared Object> {}
```

We can pass any thread-locality to `T`, as it is bounded by a `@ThreadSafe Object`. However we can only pass `@Shared` type argument to `K`.

2.3.6 Throwable and Java's Thread API

In Loci, exceptions are always thread-local. The class `java.lang.Throwable`, which is the common superclass of all errors and exceptions in Java, has been

[7] For example, a `@Shared` array of `@Local` elements.

annotated `@Local`, meaning it can only be used to create thread-local instances. If we want to use exceptions to return shared values from a deeply nested stack, you can still do so in a shared field.

The `java.lang.Thread` is a `@Shared` class and `java.lang.Runnable` is a `@Shared` interface. This is natural, as instances of the `Thread` class are always shared between two threads: the creating thread and the thread it represents [22].

2.4 Loci Through an Example

In this section, we run Loci on a simple example from the RayTracer Benchmark [3][8].

The design of RayTracer suggests us that we can annotate all the classes as `@Local`, except `Barrier` and `TournamentBarrier` which were annotated `@Shared`. The two shared classes are responsible to keep the running threads synchronized. The array of threads contains threads that start thread-local computations.

```
1  /**
2   * Barrier instances are always @Shared
3   */
4  @Shared public abstract class Barrier {
5    ...
6  }
7
8  /**
9   * RayTracer instances are always @Local
10  */
11 @Local public class RayTracer {
12   ...
13 }
14
15 /**
16  * Since this class extends RayTracer,
17  * it is implicitly @Local.
18  * This class was implementing
19  * Runnable, but since it is @Local it
20  * cannot implement a @Shared interface
21  */
22 class RayTracerRunner extends RayTracer{
23     Barrier br; //Since Barrier is @Shared, its
24                 //instances are always @Shared
25     ...
26     public void run(){...} //The core run method
27 }
```

[8] This program was one of the benchmarks that we used for evaluating Loci, for more information see Chapter 5.

```
28
29  ...
30  //Originally it was:
31  //thobjects[i] = new RayTracerRunner(i, wdt, hgt, brl);
32  //but we changed it to:
33  final int j = i;
34  thobjects[i] = new Runnable() {
35    public void run() {
36      RayTracerRunner temp = new RayTracerRunner(j,width,height,br);
37      temp.run();
38    }
39  };
40
41  th[i] = new Thread(thobjects[i]);
42  th[i].start();
```

Listing 2.2: A short example that shows how Loci performs and constrains Java programs.

The refactoring made in line 30–39 was necessary, since the `Runnable` interface is annotated `@Shared`, while we annotated `RayTracerRunner` class as `@Local`, as it extends the `RayTracer` which is `@Local`. Although in the original version the `RayTracerRunner` class implemented `Runnable` interface, we had to refactor it in order to respect the thread-locality of the `RayTracerRunner` class. In Loci a `@Local` class cannot implement a `@Shared` class, because of a scenario similar to this:

```
@Shared interface A{}
@Local class B implements A{} //In Loci, this is invalid

...
B b = new B(); //a thread-local value
A a = b; //Leak! now b is effectively shared through a
```

Because of that change, the line 31 was no longer valid, `Runnable` is no longer a parent of `RayTracerRunner`. Therefore, we needed to refactor it to make it valid.

Having looked at the Loci's logical view of memory, Loci annotations and core design and seeing an example on how to apply Loci, we now proceed to show more design decisions to give a clearer picture of the Loci system.

CHAPTER 3

DESIGN DECISIONS

> In a room full of top software designers, if two agree on the same thing, that's a majority.
>
> Bill Curtis

In the previous chapter we introduced Loci 2.0 annotations and basic design. We now proceed to see more details about the design of the system and how it differs from the old version of Loci. This chapter recaps parts of Loci 1.0 to motivate the needs for improvements. and the design decisions we made in the design of Loci 2.0.

3.1 Path of Evolution from Loci 1.0

Originally, Loci extended Java with two annotations to express thread-locality [21]. Classes annotated `@Shared` could only be used to instantiate shared values while classes annotated `@Thread`[1] could be used to instantiate both shared and thread-local values. `@Thread` was taken as the default annotation, and it could prove compatibility with legacy Java programs. Loci 1.0 was build on-top of the standard type checker presented in Java 6 and was therefore not expressive enough to cover all aspects from Java as its annotations could only appear on declarations. Furthermore, the prototype implementation of Loci 1.0 was developed as an Eclipse plugin, which never reached the maturity required to be used in production.

Based on the improvements suggested by evaluating the original Loci, we designed Loci 2.0. By making use of the upcoming annotation processor

[1] Loci 1.0 did not have `@Local` annotation, but `@Thread` instaed.

of Java 8, we were able to support annotations in more places like: object instantiations, generics and method type parameters.

There were several limitations in the design of Loci 1.0 [21]. For example, it did not allow annotations on generics and method type parameters, due to a limitation in Java annotation processor in Java 6 [13, 21, 1]. Loci was dealing with all type parameters (both on class or method level) as `@Shared`.

Another limitation with Loci 1.0 was the support of static methods. In Section 2.3.3 we mentioned that the default annotation in a static context is `@Shared`. This caused problems where static methods were used as global functions. For example in RayTracer benchmark, `Vec` class was annotated as `@Thread`, and it frequently used methods like:

```
public static Vec sub(Vec a, Vec b) {
  return new Vec(a.x - b.x, a.y - b.y, a.z - b.z);
}
```

Since `a` and `b` in the code above would be `@Shared` by default from being in a static context, no thread-local vectors are precluded from using this purely functional method [21].

Equals method was also having problems, since in a `@Thread` class, the type of `this` was `@Owner`. Therefore we could only compare objects living in the same heap(let).

To address those problems, we designed Loci 2.0, which we believe is more flexible and compatible with plain Java. We also implemented Loci 2.0 using the Checker framework [14, 11] and applied it on more than 50 000 lines of legacy Java code [8, 3]. We found that Loci 2.0 is compatible with how Java programs are written, the results are shown in Chapter 5.

3.2 Subsumption and Generics

Type annotations on generics suffer from the same problems as arrays—subsumption to `Object` allows forgetting any parameterized thread-locality at the type level. Absence of run-time type information further precludes downcasting. Thus, type arguments' thread-localities must be the same as the type parameter boundaries in Loci, unless their classes have manifest thread-locality.

```
class A<T extends @Shared Object>{}
```

The above example denotes a flexible class which takes a shared instance as a type parameter.

```
class Pair<T extends Object> {
  T fst;
  T snd;
}
```

```
Pair<Object> a; //a shared pair of thread-safe objects

Pair<@Shared Object> b; //invalid
Pair<@Local Object> c; //invalid
```

The above example takes a `@ThreadSafe` type parameter, which means `T` can hold both `@Local` and `@Shared` instances. However, there is no way of specifying a shared pair of non-`@ThreadSafe` objects except expressly bounding `T` by `@Local Object` or by `@Shared Object`. The reason behind this restriction was to prevent a leak from downcasting to happen, consider this example (which is invalid in Loci):

```
class Leaky<T extends Object>{
  T value;
}

Leaky<@ThreadSafe Object> a = ...;
Object b = a;
Leaky<@Shared Object> c =
    (Leaky<@Shared Object>) a; //Invalid, as this might lead to leak!
```

In our design, the last line is invalid as it might lead to a leak by storing a thread-local data in `a.value` and retrieving it in `c.value` as a shared data:

```
a.value = new @Local Object();
@Shared Object leak = b.value; //Leak!
```

Loci does not allow annotations on a type argument, unless:

1. It is the same as the annotation on the type argument's class, e.g. Exception is `@Local`, thus `Pair<Exception> == Pair<@Local Exception>`, or;
2. It matches the annotation on the type parameter. e.g. `Pair<Object> == Pair<@ThreadSafe Object>`.

As we mentioned above, generics suffer from the same problems as arrays when downcasting, consider this example:

```
@Local class Cell<T extends Object>{
  //equivalent to T extends @ThreadSafe Object
  T f;
}

Cell<Thread> c;
@Local Object o = c;
Cell c2 = (Cell) o;
c2.f = new Exception();
//c.f now contains @Local object, which can lead to leak!

@Shared Object = c.f; //Leak!
```

To prevent this, Loci reports a warning whenever it sees a downcasting or an assignment that goes from an instance with less type arguments to an instance with more type arguments, unless the annotations on the missing type arguments are predictable (i.e either `@Local` or `@Shared`)[2].

```
class A<T extends Object>{...}
class B<K extends Object, T extends Object> extends A<T>{...}
class C<K extends @Local Object, T extends Object> extends A<T>{
  ...
}

A<Exception> a;
B<Exception, Exception> b
     = (B<Exception, Exception>) a; //Produces a warning
C<Exception, Exception> c
     = (C<Exception, Exception>) a; //OK, K is always @Local
```

Implementing `equals` method for the classes with uncertain type parameters can be problematic, since it takes a `@ThreadSafe Object` as a parameter, which means we need to downcast the parameter to retrieve the actual data type. Fortunately there is a way to safely implement the method, by doing something similar to:

```
1  @Local public class Cell<T, K>{
2    T t;
3    K k;
4    public boolean equals(@ThreadSafe Object obj){
5      //Cell<T, K> temp = (Cell<T, K>) obj; produces a warning
6      Cell<?, ?> temp = (Cell<?, ?>) obj; //It is OK
7      return this.t.equals(temp.t) &&
8          this.k.equals.(temp.k);
9    }
10 }
```

The reason why line 6 is allowed is that in Java wildcards are read-only properties [4]. Therefore, we can never write into `K` and `T` through temp and as for reading Loci always preserves thread-locality of the two type parameters as they are both `@ThreadSafe`—they are bound by `Object` and have no explicit annotation.

3.3 Changing Class Defaults

In the first version of Loci, classes defaulted to `@Shared` as a way to give a sound Loci semantics to legacy Java code. This default, however, leads to "false negatives"—thread-local values considered shared, since most classes actually instantiate values that are safe to use as thread-locals [21]. Another unfortunate side-effect on new code is the higher annotation overhead

[2] `Cell` is considered as `Cell<Object>`

on classes to explicitly make them `@Thread`. Thus, changing the default for classes from `@Shared` to something more desirable was one of our high priority contributions (listed as C3 in Section 1.2). In Loci 2.0, classes defaults to be "flexible classes". A flexible class can be subclassed as a `@Local`, a `@Shared` or a flexible class. Also, it can be instantiated as `@Shared` and `@Local`. Thus, unless a programmer explicitly annotates a class `@Thread` or `@Shared`, it retains maximal flexibility in usage.

The big hurdle of in switching classes from `@Shared` to flexible as a default is that not all legacy Java classes are valid flexible classes. As a consequence, we had to manually flag certain classes in the JDK and the libraries of our case studies as `@Shared`, which was made using the built-in "Astub" utility of the checker framework, see Section 4.2.

This change helped Loci 2.0 to require fewer annotations than the previous version. This can be noticed in the comparison between annotating RayTracer in Loci 1.0 and 2.0 (Chapter 5). Our version needed less annotations (22 vs 15).

3.4 Locality Polymorphic Methods

In Loci, code in a static context is considered to operate in the shared heap since static fields and methods and classes are global entities that are accessible by any thread. Consider the following class:

```
@Local class Foo {
  static Object f;
  static {
    @Shared Object x = new Object();
    f = x; // OK, f is implicitly @Shared
  }
  static Object id(Object o1) {
    return o1;
  }
  static Foo fooId(Foo o2) {
    return o2;
  }
}
```

Listing 3.1: Shows how types within a static context are treated in Loci.

Instances of flexible classes are implicitly `@Shared` in static contexts. Therefore, the `id(Object o1)` method in the Listing 3.1 accepts a `@Shared` parameter and returns a `@Shared` type. Typing classic functions such as `id` or `max` are long-standing problems for Loci's designers. In Loci 1.0 there was no good way for implementing it, as it lacked support for generics, the only way was to have two identical methods with different thread-locality:

```
public @Thread Object threadId(@Thread Object a){return a;}
public @Shared Object sharedId(@Shared Object a){return a;}
```

which of course destroys polymorphism and forces code duplication.

Using generics, Loci 2.0 allows merging the above two methods in a straightforward fashion:

```
static <T extends @ThreadSafe Object> T id(T o) { return o; }
```

Sadly, however, the above strategy is not problem-free. Notably, the method takes a `@ThreadSafe` parameter and returns a `@ThreadSafe` type, which means that the method effectively loses thread-locality. It would be desirable to retain it, i.e., pass in an argument with a certain thread-locality and get a type with the same thread-locality back. For example when performing the `binarySearch` using `Arrays` class, we pass an array with a certain thread-locality and expect an element with the same thread-locality. Therefore, we introduced an existential annotation called `@X` that can only appear on method type parameters. In this setting, the existential annotation means that the thread-locality is unknown, and must therefore be treated conservatively (same constraints as `@ThreadSafe`). Using existentials, the `id()` method can be re-written as follows:

```
static <@X T extends Object> T id(T o) { return o; }
```

When a method is called, the existential parameters are bound implicitly to the thread-locality of the arguments. For example, if `id()` is called with a `@Shared` parameter, `@X` is bound to `@Shared` and consequently, the return type is also `@Shared`. For `@Local` arguments, we get a `@Local` type back in the same way. `@X` annotations can be given unique identifiers, to support methods with several parameters with different unknown annotations.

```
<@X(1) T extends Object, @X(2) B extends Object> T m(T a, B b){
  return b; // Illegal @X(2) != @X(1)
  return a; // OK
}
```

Listing 3.2: A method with more than one `@X` annotations

In the above example, `B` and `T` may or may not have the same thread-locality. Therefore, Loci disallows assignment between them. In the obvious way, we can also require two parameters to have the same thread-locality with possibly different types:

```
<@X T extends Object, @X B extends Object> @X Object m(T o1, B o2) {
  if (random) return o1; else return o2; // OK
}
```

Listing 3.3: Two method type parameters that have the same thread-locality

3.5 Annotating Common Methods

Every valid Java class has the methods `equals` and `clone`, either through defining them itself, or by inheriting them from `Object`. It is important

that Loci can be used with these two methods, especially `equals` as many programming patterns in Java rely on structural equality tests.

In this section, we briefly discuss the typing of these methods in Loci 2.0[3].

3.5.1 The Equals Method

Loci 1.0 did not have a flexible annotation like `@ThreadSafe` and therefore, could only allow equality tests between two shared objects or two thread-local objects [21]. This is limiting, as lifting equality over identity makes it possible for a shared object to be equivalent to a thread-local one, and vice versa.

With the addition of the `@ThreadSafe` annotation in Loci 2.0, the equals method can be typed in a more flexible way which allows test between objects with different thread-locality. (This is item C4 in our contributions list, in Section 1.2.) The resulting signature in `Object` is:

```
public boolean equals(@ThreadSafe Object that){...}
```

Notably, due to the defaults, the `@ThreadSafe` annotation is not even necessary (it will be elaborated in as the default), and the flexible equality comparison falls out automatically.

3.5.2 The Clone Method

The `clone()` method's return type is effectively `@Owner` which means every instance clones to an instance with the same thread locality of the original, and for having cross thread-locality cloning we have to have additional methods[4].

It should be safe to assign a `@Local` cloned instance to a `@Shared` variable or vice versa, although the type system currently prevents thisLoci. A properly cloned instance should be a freshly created instance which has not been assigned to any external variable or field. Therefore, assigning the result of a *proper, deep clone* to either a `@Local` or a `@Shared` variable does not cause a thread-locality violations. However, since clone methods in Java must be written manually by the programmer, there are no guarantees as to what is returned form a clone method, and consequently, Loci cannot rely on their proper implementation.

We presented the design of Loci 2.0, now it is time to proceed to presenting the Loci plugin, to give a clearer view of the Loci system.

[3] Although the `Object` class has 11 methods, we need to care about `equals` and `clone` methods only, because these two methods are the only methods in this class with types from flexible classes in their signatures.

[4] They are possibly identical to the actual `clone()` method with different annotations.

CHAPTER 4

THE LOCI TOOL

> It is better to do the right problem the wrong way than the wrong problem the right way.
>
> Richard Hamming

The previous chapters gave a detailed introduction to thread-locality and the design of our system, now is the time to see Loci in action by introducing the compiler plugin that we have implemented. The chapter starts with a brief introduction to JSR 308 and the Checker framework and then presents the Loci compiler plugin and the known bugs and limitations in the current release of Loci. This chapter is concerned with our fifth contribution in the list in Section 1.2.

4.1 JSR 308

Java has a limited support for metadata annotations since version 5. The annotation system is however not expressive enough to allow annotation on all Java features. For example, it does not support generics or annotations on class instantiations. To overcome this limitation, JSR 308—"annotations anywhere"—has been suggested and approved. JSR 308 adds support for annotation on, for example, generic type arguments (e.g. `List<@LocalObject>`) [6]. This JSR is planned to be part of OpenJDK 8[1]. There is an early access release of the type annotations compiler which can be used with the current version of Java (or even OpenJDK 7) and it is freely available for downloading.

[1] It was planned to be part of OpenJDK 7 [13] but was postponed to OpenJDK 8 [5].

Using the JSR 308 compiler does not break backward compatibility of Java code which makes it possible to gain the benefits of type annotations while working with people who are using other versions of Java, such as Java 6, 5 or even 4.

We decided to use the JSR 308 compiler for implementing Loci rather than relying on a specific front end or other syntactic extension to Java which would require additions to developers' Java tool chains.

As an aside, Java Specification Requests, are the actual descriptions of proposed and final specifications for the Java platform. JSRs are reviewed by the Java Community Process and made public before a final release of a specification. The Java Community Process is a formalized process established in 1998 that allows interested parties to get involved in the definition of future versions and features of the Java platform.

4.2 The Checker Framework

The Checker framework is a framework based on the JSR 308 type annotations compiler. It provides several *checkers* that can be plugged into the compiler on-demand, like the "Nullness Checker" for non-null types and the Javari Checker for read-only references among others [14, 11]. In addition to a set of default checkers the framework also enables developers to build custom checkers on top of the framework quite easily. The implementation of Loci consists of three parts, the thread-locality checker, the annotator and the astub utility, described below.

The Thread-Locality Checker

The Checker framework provides a base visitor class called `BaseTypeVisitor`, which follows the visitor design pattern and type-checks each node of a source files's Abstract Syntax Tree (AST) [14]. This class reports type violations in terms of error messages or warnings depending on the desired semantics. This is a key class of the type checking functionality of the system and is intended to be extended by plugin developers. Indeed, the central class `LociVisitor` in the Loci implementation extends `BaseTypeVisitor` and provides specialisations for most of its methods to implement the Loci semantics.

The Annotator

The Checker framework provides three different classes to implicitly annotate a source code. Loci extends them as follows:

LociAnnotatedTypeFactory extends `AnnotatedTypeFactory` to process `@Owner` and `@X` annotations.

LociTreeAnnotator extends `TreeAnnotator` to implicitly annotate the types upon declarations; and

LociTypeAnnotator extends `TypeAnnotator` to implicitly annotate the types in other places.

The last two classes are key components in Loci to lower the needed annotations overhead, because these two enable Loci to infer annotations when possible.

The Astub Utility

Astub is a utility program for annotating the Java classes without changing the actual class files, which is very handy for annotating core classes of the system which are not accessible for security reasons. The tool reads the annotations from a plain-text file which contains the Java class interface declarations with annotations that are "overlay-ed" on the actual classes. In Loci we have used Astub to flag shared classes in the JavaSE library by annotating them as `@Shared`, which is its intended use. A shortcoming of the Astub utility is its lack of support for annotations on generics, method type parameters and inner classes. As was discussed Section 5.1.1, this limitation caused a few problems for the Loci tool.

Loci processes one AST node at a time, the order depends on the `javac` compiler. It first checks whether the node is already annotated. If not, Loci annotates the unannotated nodes when possible by either inferring the proper annotation from the context or by using the Astub utility program. Once the node is annotated, Loci tries to process `@Owner` and `@X` annotations, as mentioned in 2.3.1 and 2.3.3 respectively. Last, Loci checks the node for thread-locality violations and prints error messages upon violations. This process is shown in detail in Figure 4.1.

4.3 Installing and Running Loci

Using the Checker framework and following the JSR 308 specifications we created a checker as a `javac` plugin. Our Loci plugin is free and open source, and available online: `http://www.it.uu.se/research/upmarc/loci`. The tool is easily installable with few dependencies, and as a pure Java command line tool, is Operating System and IDE independent.

Installing Loci is as easy as putting the distributed JAR file in your `classpath` environment variable once you have correctly installed the JSR 308 type annotator compiler. Loci can also be used as a Maven2 plugin, with ANT build script or different IDEs [22]. To apply Loci on your source code you have to provide Loci as a compiler processor to `javac`:

```
$ javac -processor loci.LociChecker *.java
```

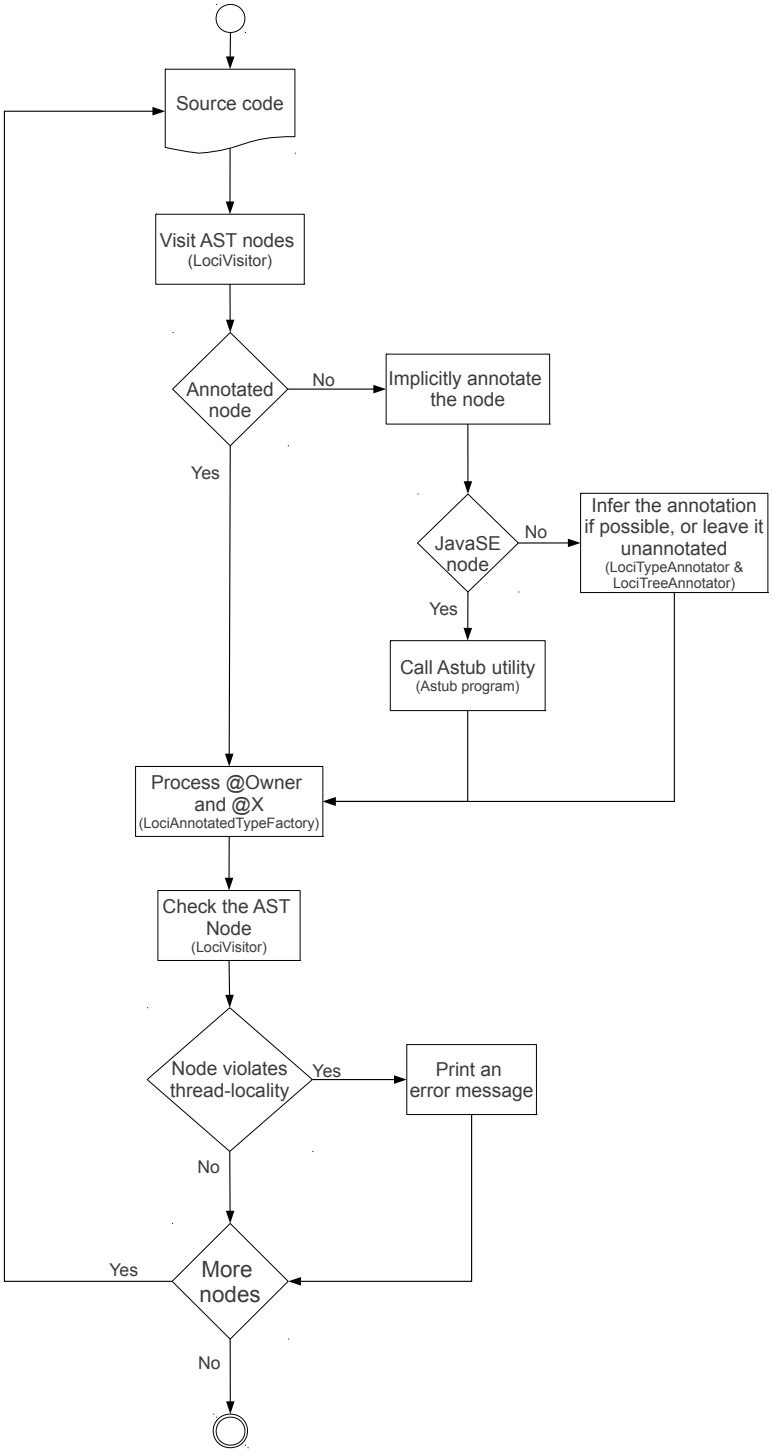

Figure 4.1: How Loci processes and checks source code.

Loci is designed in a way to be compatible with ordinary Java programs, and programs written for Loci can also be run and compiled with ordinary JVM and `javac`. Note that it is possible to compile and run programs written for Loci with Java 5 and up, if the annotations were put inside comments, like: `/*@Local*/ Object b`.

4.4 Known Bugs and Limitations

At the time of writing, Loci still has a small number of known bugs. They stem either from the Loci code base itself or from the Checker framework and affecting Loci. In this section we explain them, and offer a way to avoid them where possible.

4.4.1 Implementing Parameterized Interfaces

When implementing a parameterized type (e.g. `Comparable`), if we provide the type arguments, we need to provide their annotations too, otherwise Loci rejects the code. For example, this code produces an error:

```
1 @Local class A {}
2 class B extends A implements Comparable<B> {
3   public Comparable getValue() {
4     return new Test();
5   }
6 }
```

Loci rejects the line number 5 in the above example with the following error message:

```
Incompatible types:
   return new Test();
          ^
found: B implements Comparable<@ThreadSafe B>
required: Comparable<B>
```

The reason for this bug is that Loci tries to infer the annotations on unannotated classes (and interfaces) from the `extends` and `implements` clause. In the above example, when Loci looks at the `implements` clause, it does not know the thread-locality of `B` yet, which happens to be the type argument for `Comparable` too. The error goes away, however, if we supply an explicit annotation to the type argument of the parametric superclass:

```
@Local class A {}

class B extends A implements Comparable<@Local B> {
  public Comparable getValue() {
    return new Test(); // OK
  }
}
```

4.4.2 Anonymous Classes Need Explicit Annotations

Anonymous classes that are implicitly `@Shared` or `@Local` will provide false error reports that go away if the correct annotation is provided explicitly. For example:

```
1  abstract class A { void m(); }
2
3  @Shared class B {
4    Object method() {
5      return null;
6    }
7
8    void m1(){
9      A a = new @Shared A() {
10       public void m() {
11         Object b = method();
12       }
13     }
14   }
15 }
```

At line number 9 above, we need to put the `@Shared` annotation explicitly, even though Loci should elaborate it in automatically as the line is inside the scope of a `@Shared` class. If not, the line number 11 is rejected with the following error:

```
Incompatible types:
    Object b = method();
           ^
found: @Owner Object
required: @Shared Object
```

The reason for this bug is that the checker framework starts to check the typing of line number 9 before inferring the annotation of the anonymous class.

4.4.3 `@Owner` Causes Problems in Subclassing

When a non-flexible class inherits an `@Owner` field, return type or parameter from its flexible parent, Loci fails to convert the `@Owner` to match the annotation of the non-flexible class. Suppose we have these two classes:

```
1  class A { // A flexible class
2    void method(Object b) {...} // The annotation on b is @Owner
3  }
4  @Shared class B extends A {
5    void method(Object b) { // b is implicitly @Shared
6      super.method(b);
7    }
8  }
```

On line number 7, Loci reports an error saying:

```
Incompatible types:
    super.method(b);
                 ^

found: @Shared Object
required: @Owner Object
```

We believe that this is a bug inherited from the Checker framework.

4.4.4 Miscalculating Arrays of Type Arguments

As we mentioned in Section 2.3.5, Loci arrays always have the same thread-locality as their elements. Consider the following:

```
1  class B<T> {
2    T[] method() {...}
3  }
4
5  B<Exception> b = new B<Exception>();
6  Exception[] array = b.method();
```

Line number 6 produces an error:

```
Incompatible types:
    Exception[] array = b.method();
                ^

found: @Local Exception @ThreadSafe []
required: @Local Exception @Local []
```

Processing type parameters in the Checker framework is completely automatic and there is no way to program it. Therefore, when Loci reaches the line number 6, the Checker framework substitues the annotation on the return type of the `method()` to match the annotation given in the line 5. Unfortuantely, Loci cannot force the checker framework to change both the annotation on the array type and its elements during this process, and thus the problem occurs.

4.4.5 Incorrect Inference of Method Type Arguments

This is a bug from the Checker framework, and it is already patched but have not been pushed to a stable version yet. Once the patch is released, Loci will work correctly in this respect[2].

If a method type parameter which has `@X` annotation appears on both method parameter and return type, Loci fails to find the correct thread-locality:

```
1  class A{
2    public static <@X T, @X(2) K> T badMethod(T t, K k){...}
```

[2]Bug report: http://code.google.com/p/checker-framework/issues/detail?id=93

```
3 }
4
5 @Local Object a = ...;
6 @Shared Object b = ...;
7
8 @Local Object c = A.badMethod(a, b);
```

Although, the line 8 should be accepted but Loci fails to accept it, complaining:

```
Incompatible types:
   @Local Object c = A.badMethod(a, b);
                     ^
found: @Local Object
required: @ThreadSafe Object
```

Having looked at the Loci tool and its current known bugs, we proceed to see Loci in action by presenting our results in running Loci on some well-known Java benchmarks.

CHAPTER 5

EVALUATING LOCI

> An algorithm must be seen to be believed.
>
> Donald Knuth

The previous chapters presented the design of Loci 2.0 and the Loci tool plugin. In this chapter we evaluate our work by running it on several well-known Java benchmarks from Java Grande [3] and DaCapo [8] benchmark suites. Then we share the results and conclude them.

5.1 Benchmarking Loci

To evaluate Loci's type system design, we annotated a number of Java programs from both Java Grande [3] and Dacapo [8] benchmarks as well as the Lucene Search library from Apache [2]. In total we annotated 3 programs and more than 50 000 LOC. Our policy was to select heavily multi-threaded programs which are used for real-world benchmarking. We focused on the the pre-Java 5 programs, in our evaluation process, to "guaranty" that Loci is compatible with them.

The largest code base we annotated was the Lucene searching library version 2.4.1 which consists of 367 source files many of which defines more than a single class or interface. Due to the large code base we decided to annotate only the parts that are reachable from the `IndexReader` class which is needed for the Lusearch benchmark from the DaCapo benchmark suite [8].

To explain the evaluation process, we take an excerpt from the RayTracer benchmark, and present the same excerpt before benchmarking afterwards.

The original code looked thus:

```
/**
 * Barrier instances are always @Shared
 */
@Shared public abstract class Barrier {
  ...
}

/**
 * RayTracer instances are always @Local
 */
@Local public class RayTracer {
  ...
}

/**
 * Since this class extends RayTracer,
 * it is implicitly @Local.
 * This class was implementing
 * Runnable, but since it is @Local it
 * cannot implement a @Shared interface
 */
class RayTracerRunner extends RayTracer{
    Barrier br; //Since Barrier is @Shared, its
                //instances are always @Shared
    ...
    public void run(){...} //The core run method
}

...
//Originally it was:
//thobjects[i] = new RayTracerRunner(i, width, height, br);
//but we changed it to:
final int j = i; // This step is needed to access the index in
                 // the ''thobjects[i]'' scope.
thobjects[i] = new Runnable() {
  public void run() {
    RayTracerRunner temp = new RayTracerRunner(j,width,height,br);
    temp.run();
  }
};

th[i] = new Thread(thobjects[i]);
th[i].start();
```

Listing 5.1: The refactored Raytracer program.

```
/**
 * This class is responsible to keep the running
 * threads synchronized.
 */
public abstract class Barrier {
  ...
}

/**
 * This class creates an array of RayTracerRunner.
 * Then, starts the array items (threads).
 */
public class RayTracer {
  ...
}

/**
 * The run method of this class, traces the different
 * objects in the ''world'' and produces the image.
 */
class RayTracerRunner extends RayTracer implements Runnable{
    Barrier br;
    ...
    public void run(){...}
}

...
thobjects[i] = new RayTracerRunner(i,width,height,br);
th[i] = new Thread(thobjects[i]);
th[i].start();
```

Listing 5.2: An excerpt of the RayTracer benchmark.

If we look at the original code from RayTracer Benchmark and the one that we have annotated (and refactored) we can notice one major difference. In the original version RayTracerRunner implements Runnable but in the modified version it does not. The reason we need to do this is, RayTracerRunner is `@Local` by design and Runnable is `@Shared` by design, so the implementing Runnable would not be sound and result in a compile error.

After naively annotating the program to reflect the design intents, Loci (correctly) rejected it, requiring us to refactor it slightly. This is the only refactoring we have done in our benchmarking on any of the programs.

The Lucene Search Library

We annotated $\approx$ 46 KLOC of the Lucene library. This corresponds to two thirds of the entire library. Our impression of this undertaking were that most of the time *a single annotation on the class level is enough to express the intended behaviour with respect to thread-locality.* Annotations on fields,

Place	`@Shared`	`@Local`	`@ThreadSafe`	**Total**
Class and Interface	23	45	-	68
Return	13	20	27	60
Parameter	5	73	27	105
Variable	10	110	52	172
				Annotation/KLOC
Total	51	248	106	≈ 9
				NOC/NOI
				243 classes and 13 interfaces
				LOC
				45853 lines of code

Table 5.1: The result of annotating the Lucene Search Library.

variables, etc. were much less frequent. We believe that this suggests that our design decisions help in reducing of annotations needed per line of code with the desired semantics. In average we needed ≈ 9 annotations/KLOC. We believe that this is a very small addition for avoiding thread-locality violations. Table 5.1 shows the annotations we have added for Lucene Search library in detail.

For pragmatic reasons, Loci allows methods to be annotated as `@ManuallyVerified`. The annotation is to be used sparingly, if at all. `@ManuallyVerified` forces Loci to skip checking the body of a method. However, Loci still checks the return type and the parameters of the method. Due to (mostly) heavy use of pre-Java 5 collections, we were forced to use `@ManuallyVerified` heavily in this library. This annotation is meant to be used very carefully as it constitutes a backdoor that can cause unsoundness in the system. In the Loci implementation, it is used only in a single place in the `ThreadLocal` as this class is part of Loci's core semantics.

The use of `@ManuallyVerified` in Lucene library was due to existing bugs either in Loci or in the "Astub" utility (see Section 5.1.1). In total we had 22 occurrences of this annotation. 15 of them were because of existing bugs in Loci: 2 of them because of the bug mentioned in Section 4.4.5, another 2 because of the bug mentioned in Section 4.4.3 and the other 11 were because of the bug mentioned in Section 4.4.1. Fixing these bugs eliminates the need of these `@ManuallyVerified` annotations.

Our experience with Lucene library further guided our design, for instance: in generics, annotations on type parameters were the same as the types, unless they were explicitly annotated. But this is no longer correct, as Loci can have mixed thread-locality generics, more on this can be found in Section 2.3.5, or before it was valid to write into an `@Owner` field of a

Place	`@Local`	`@Shared`	**Total**
Class	10	1	11
Variable	2	2	4
			Annotation/KLOC
Total	12	3	10.45
			NOC/NOI
			15 classes
			LOC
			1424 lines of code

Table 5.2: The result of annotating the RayTracer benchmark.

`@ThreadSafe` instance, but this is no longer allowed due to the reasons mentioned in Section 2.3.4. It also suggested us a huge amount of bug fixes. And we implemented most of the design suggestions we gained from this library in the initial version of the Loci tool release.

The RayTracer Benchmark

The ray tracer benchmark is a part of the Java Grande benchmark suite. It is a multi-threaded ray tracing program of 1436 LOC. Capturing its thread-local behaviour was straightforward and only required a minor refactoring. The ray tracer consists of 17 classes needing only 15 annotations in total. Of these annotations, 11 were on class-level, Table 5.2 shows the annotations we have added for the RayTracer benchmark in detail.

In the initial Loci work, Wrigstad et al. benchmarked the RayTracer program [21], and their result was quite similar to ours. The main differences between the two are due to changed defaults; in Loci 1.0 classes were implicitly shared unless they were explicitly annotated `@Thread`, but in Loci 2.0 classes default to flexible. Overall, Loci 2.0 required fewer annotations than Loci 1.0 which needed 21 annotations. On class level, Loci 2.0 needs two shared annotations whereas Loci 1.0 needs none. Another difference between the two was that Loci 2.0 needed annotations on classes (1 `@Shared` and 10 `@Local`) and variables (2 `@Shared` and 2 `@Local`) only, while Loci 1.0 needed annotations on classes (16 `@Thread`), fields (1 `@Shared`), parameters (1 `@Shared`), return types (1 `@Shared`) and variables (1 `@Shared` and 1 `@Thread`). As a side note, in our RayTracer benchmark, we could have safely made the classes annotated `@Thread` in Loci 1.0 flexible classes, but as the use of these classes suggests that they are strictly thread-local, we preferred to annotate them `@Local`.

Place	`@Local`	`@Shared`	`@ThreadSafe`	**Total**
Class	1	1	_	2
Variable	0	0	3	3
Parameter	0	0	1	1
				Annotation/KLOC
Total	1	1	4	18.69
				NOC/NOI
				4 classes
				LOC
				321 lines of code

Table 5.3: The result of annotating the Lusearch benchmark.

The Lusearch Benchmark

The last program we annotated was the Lusearch benchmark from the DaCapo benchmark suite [8] of only 321 LOC. Sadly the results from this class are not valid for the latest release of Loci, as this class heavily depends on Lucene library which has not been updated due to the large code base. The results of this class can be found in Table 5.3.

5.1.1 Observations

Unlike Lusearch and RayTracer, annotating Lucene library was rather difficult, because the large code base was a bit hard to analyze and understand. We needed to revise and re-annotate already annotated parts of the library several times. This was important to keep the different parts of the library compatible with each other, so they altogether pass Loci. This is not very surprising, as it is at least an order of magnitude bigger than the other benchmark programs.

To make our benchmarking accurate, we needed to use an annotated version of JavaSE, therefore, we annotated the JavaSE 6 classes based on the results of a simple program analysis carried out by Filip Pizlo that checked whether a class could safely be used as a thread-local value. The annotation was automated by a small script that put a `@Shared` annotation on all values that could not be safely used as thread-local values. This crude level of annotations was manually complemented in some cases, mostly utility methods in utility classes such as `Arrays`, `Collections`, and `System`. Overall we manually annotated 38 methods from 14 different classes. Our annotated version of JDK 6 classes are already in the Loci tool distribution and can be used with any other projects that use Loci.

Limitations in "Astub" Utility Program

The utility program that we created for annotating JDK classes was relying on a utility program called "Astub" from the Checker framework. Unfortunately the "Astub" utility program lacks some features, for example, it does not allow annotations on inner-classes and type parameters. This dramatically limited our capabilities of externally adding annotations to JDK classes, which forced us to use `@ManuallyVerified` instead of annotating the proper classes or type parameters when needed. This problem would go away if we annotate the actual source files of the Java classes.

5.2 Conclusions

In conclusion Loci seems to be a useful tool for programmers to write correct programs with respect to thread-locality. Furthermore, Loci seems to be compatible with how Java programs are written and only require a relatively small number of extra annotations in the program code.

It should be noted that the major part of the evaluation was conducted before "all" of the design bugs were weeded out. In the RayTracer and Lusearch benchmarks we could easily and accurately analyze the program and discover the thread-locality manually, however that was not the case in the Lucene Search library due to the large code base. This may have negative effect on the accuracy of our evaluation process.

CHAPTER 6

CONCLUSION

> We can only see a short distance ahead, but we can see plenty there that needs to be done.
>
> Alan Turing

The previous chapters presented our extensions to Loci system, and the tool that we have designed and how the tool interacts with real-world Java programs. In this final chapter we conclude our work and suggest possible improvements.

6.1 Results

In this thesis we extended a simple type system for expressing thread-local data in Java and Java-like languages. We introduced Loci 2.0 and showed its semantics through a number of examples and discussed design decisions. We also discussed the command-line tool that implements the type system which was used in the practical evaluation. The tool was constructed for this thesis and is available online at `http://java.net/projects/loci/downloads/directory/Loci-0.1`.

We believe that the number of annotations/LOC that Loci requires is reasonably low and that the buy-in for "porting" legacy Java code is relatively cheap. Also, using Loci required no substantial rewrites and only a small number of simple refactorings. In its current design, Loci supports all Java constructs, but there is still room for improvement like adding a better support for cloning or moving thread-local values across threads to improve the handling of producer-consumer scenarios. Notably, no such scenarios were found in the evaluation process.

Another interesting notion would be to support mediating a value from `@Local` to `@Shared` or the other way around.

The core idea behind keeping annotation overhead at a minimum in Loci is the class-level annotations and the choice of defaults. We are happy to report that in most cases, a single class-level annotation was enough to capture the desired semantics. Also, `@Local` was more used than `@Shared` on class level in our practical evaluation, which reflects our initial understandings and also the results from Wrigstad et al. [21] which says that most of the classes can be annotated `@Thread`[1].

With respect to return types, `@ThreadSafe` was the most used annotation, `@Local` the second and `@Shared` the last. The high number of `@ThreadSafe` annotations was because the code was written for Java 1.4 and not using generics, and because of the limitation in the "Astub" utility program in the Checker framework which is mentioned in n Section 5.1.1.

As for parameters `@Local` was used far more than `@ThreadSafe` which was used more than `@Shared`. This is encouraging as most classes are not `@Shared`. On the variable level, the ranking was `@Local`, `@ThreadSafe`, `@Shared`. `@Shared` was only used 12 times (in relation to `@Local`, which was used 73 times, i.e., 6 times as often).

6.2 Future work

Loci still has room for improvement. Transferring values across threads without copying is an important next goal. This might be possible by applying a Scala-like uniqueness capabilities [16] or external uniqueness [10]. The major problem is finding a system that imposes as few extra annotations as possible on the programmer.

Modulo objects which incorporate global variables in its structure, cloning an object should return a reference which would be safe to treat either as `@Local` or `@Shared`. Currently, objects that should be clonable to both `@Local` and `@Shared` needs two different methods, which are completely the same modulo annotations. Introducing some kind of unique or free value might be beneficial both for flexible cloning and for transfer.

Contributing to Loci

The Loci command-line tool is open source and anyone who wishes can grab the source code and study it. In fact we welcome all contributions and suggestions. For more information and the latest updates please visit this website: `http://www.it.uu.se/research/upmarc/loci`.

[1] `@Thread` in Wrigstad et al. [21] on classes is identical to flexible classes in Loci 2.0.

BIBLIOGRAPHY

[1] Annotation processing tool (apt). [Online]. Available: http://download.oracle.com/javase/6/docs/technotes/guides/apt/index.html

[2] Apache Lucene library. [Online]. Available: http://lucene.apache.org/java/docs/index.html

[3] The Java Grande Forum Multi-threaded Benchmarks. [Online]. Available: http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/contents.html

[4] Java wildcards. [Online]. Available: http://download.oracle.com/javase/tutorial/extra/generics/wildcards.html

[5] JDK 7 Features. [Online]. Available: http://www.openjdk.java.net/projects/jdk7/features/

[6] Locations for annotations in JSR308. [Online]. Available: http://types.cs.washington.edu/jsr308/specification/java-annotation-design.html#locations-for-annotations

[7] ThreadLocal API. [Online]. Available: http://download.oracle.com/javase/6/docs/api/java/lang/ThreadLocal.html

[8] S. M. Blackburn, R. Garner, C. Hoffman, K. S. Khan, A. M.and McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, M. Hosking, A.and Jump, H. Lee, J. E. B. Moss, D. Phansalkar, A.and Stefanović, T. VanDrunen, and B. von Dincklage, D.and Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[9] G. Bracha, "Pluggable type systems," in *OOPSLA04 Workshop on Revival of Dynamic Languages*, Vancouver, Canada, 2004.

[10] D. Clarke and T. Wrigstad, "External uniqueness is unique enough," in *European Conference for Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2003, pp. 176–200.

[11] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. Schiller, "Building and using pluggable type-checkers," in *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, Waikiki, Hawaii, USA, May 25–27, 2011.

[12] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald, "Thread-local heaps for java," in *Proceedings of the 3rd international symposium on Memory management*, ser. ISMM '02. New York, NY, USA: ACM, 2002, pp. 76–87. [Online]. Available: http://doi.acm.org/10.1145/512429.512439

[13] M. Ernst. (2007) Annotations on Java types, JSR308. [Online]. Available: http://www.jcp.org/en/jsr/proposalDetails?id=308

[14] ——. (2008–present) The checker framework: Custom pluggable types for Java. [Online]. Available: http://types.cs.washington.edu/checker-framework/

[15] D. Flanagan, *Java In A Nutshell, 5th Edition*. O'Reilly Media, Inc., 2005.

[16] P. Haller and M. Odersky, "Capabilities for Uniqueness and Borrowing," Tech. Rep., 2009. [Online]. Available: http://lamp.epfl.ch/~phaller/capabilities.html

[17] Y. Lu, J. Potter, and J. Xue, "Ownership downgrading for ownership types," in *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, ser. APLAS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 144–160. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10672-9_12

[18] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble, "JavaCOP: Declarative pluggable types for Java," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 4:1–4:37, February 2010. [Online]. Available: http://doi.acm.org/10.1145/1667048.1667049

[19] B. Steensgaard, "Thread-specific heaps for multi-threaded programs," in *Proceedings of the 2nd international symposium on Memory management*, ser. ISMM '00. New York, NY, USA: ACM, 2000, pp. 18–24. [Online]. Available: http://doi.acm.org/10.1145/362422.362432

[20] B. Weissman, "Performance counters and state sharing annotations: a unified approach to thread locality," *SIGPLAN Not.*, vol. 33, pp. 127–138, October 1998. [Online]. Available: http://doi.acm.org/10.1145/291006.291035

[21] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek, "Loci: Simple thread-locality for Java," in *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 445–469. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_21

[22] T. Wrigstad and A. Sherwany, *Loci 0.1 Manual*, 2011. [Online]. Available: http://loci.java.net/manual