Loci: Simple Thread-Locality for Java

Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, Jan Vitek

Computer Science Dept.
Purdue University

Abstract. This paper presents a simple type system for thread-local data in Java. Classes and types are annotated to express thread-locality and unintended leaks are detected at compile-time. The system, called Loci, is minimal, modular and compatible with legacy code. The only change to the language is the addition of two new metadata annotations. We implemented Loci as an Eclipse plug-in and used it to evaluate our design on a number of benchmarks. We found that Loci is compatible with how Java programs are written and that the annotation overhead is light thanks to a judicious choice of defaults.

1 Introduction

Statically determining whether part of a computation takes place solely inside a single thread is desirable for several reasons. Not only does it simplify reasoning, but it enables optimizations that are only possible in sequential code, for example, to improve the performance of automatic memory management algorithms [17, 32] or remove unnecessary synchronization [2, 1]. Java has supported *thread-local fields* with the `ThreadLocal` class since version 1.2 of the language. Using this API, each thread can have its own copy of a field and use that in a race-free manner. Another example of use is the detection of deadlock and shutdowns in a CORBA implementation [25].

When used with simple or immutable data types, the `ThreadLocal` API offers sufficient protection. However, when used with mutable complex data types the safety offered by the `ThreadLocal` API can be likened to that of name-based encapsulation: the *field* is guaranteed to be thread-local, but not its *contents*. For example, a reference obtained from a thread-local field can subsequently be shared across threads, which may violate thread-locality assumptions elsewhere in the program. This also means that compilers can not rely on thread-locality in their optimizations.

In this paper, we propose a simple, statically checkable set of annotations that lets programmers express thread-locality. Our system, Loci, was designed with Java in mind, but should apply to Java-like languages with only a modicum of changes. We extend Java with two annotations: `@Thread`, to denote potentially thread-local objects and `@Shared` to denote shared objects. Classes that do not leak **`this`** to shared fields can be marked `@Thread` to denote that they are safe to use for thread-local computation. Annotations on fields, parameters, local variables and method returns are then used to express thread-locality of objects in the program. Loci statically verifies that a program's thread-local behavior corresponds to the programmer's intentions and thus enforces proper use of Java's `ThreadLocal` API. Experiments with Loci on ≥45 000 lines of Java code validate our design and suggest improvements.

The design of Loci was driven by a quest for simplicity and practical applicability. To minimize syntactic overhead, and increase reusability of code and libraries, Loci uses a default annotation system on types controlled by class-level annotations. In many cases, a single class-level annotation is necessary to make a class suitable for use in thread-local computation. Furthermore, the defaults have been chosen so that all existing Java programs are valid Loci programs. Loci is an ownership types system that uses *threads* as owners, instead of objects, similar in spirit to [30, 14]. This paper makes the following contributions:

1. It proposes Loci, a simple annotation system for Java-like languages that can statically express and enforce proper use of thread-local data and integrates reasonably well with legacy code, specifically, all existing Java programs have a valid Loci semantics.
2. An implementation of Loci in an Eclipse plug-in which integrates error reporting with the Eclipse IDE and performs bytecode-level rewriting of thread-local field accesses and thread-local methods.
3. Reports on experimental results from annotating classes in existing Java programs. We have refactored some benchmark programs by hand and, in parallel, we have implemented an inference algorithm as well as a dynamic tracking algorithm.

The main motivation for our work is to statically enforce thread-locality in Java. Loci allows programmers to declare their intentions with respect to thread-locality and checks statically that those intentions are never violated. The knowledge that some of the data manipulated by a system is guaranteed to be race-free is a big help for programmers as they need not worry about concurrency control for those parts of the system.

There are other potential benefits to thread-locality that we intend to explore. For starters, we believe that thread-locality information can be used to improve performance. Trivially, thread-local objects are free from data races and no locks need to be acquired for such objects. Thread-locality also has positive effects on garbage collection. For example, collecting thread-local data can be done in parallel without synchronization. In a reference counted collector, reference counts on thread-local objects can be modified directly without a compare-and-swap. Previous work [17] shows overall speedups of 50% when using thread-local heaps in Java. As a last example, thread-locality could be used to cache field reads for local manipulation in methods, before written back.

Another interesting future application of thread-locality lies in real-time computing. Run-time facilities used to execute Java programs typically have a very fast path that speculatively assumes properties that would be implicit with thread locality. For example, thin locks [3] and biased locks [29] assume a lack of contention. While this results in good performance for systems in which only the overall throughput is important, it is of no use to systems in which the worst-case performance is the more interesting property. Thread-locality can aid the worst-case analysis of real-time programs, by assuring the developer that the slow-path will never be taken. Typical Java programs use locking in the form of the `synchronized` statement freely under the assumption that it is cheap. Unfortunately, locking is only cheap when it is uncontended. Proving this with a static analysis is hard, but can be made easy for a large subset of the program by using Loci. While real-time garbage collection is gaining acceptance, its performance is still lagging. Using thread-local heaps, or similar variants thereof [27, 30], increases both mutator utilization and scalability. Existing systems accomplish this with a combination of run-time checks, analyzes performed at run-time on object graphs, and extremely strict confined types systems. Loci could be used to achieve a similar effect with more expressive power and less work on the part of the programmer.

2 Informal Introduction to Loci

We now informally describe the Loci system, its logical run-time view of the heap and the annotation system.

2.1 Example

As first example, Fig. 1 shows Loci preventing a supposedly thread-local variable from a thread's `run` method to be leaked. The class `Leaky` is a thread with a field `unsafe` of type `String[]`. The `run()` method stores a reference to an object that was intended to be thread-local into that field, thus making it possible for another thread to read the field and perform concurrent updates on the array.

In many cases, the only variable that needs to be explicitly annotated in Loci is the root variable on the bottom stack-frame in the thread's `run`-method. The Loci class `Safe` is a thread where the field `unsafe` has been marked `@Shared`. This means that it is a field that can be read by multiple threads. In the method `run`, the `local` variable is marked `@Thread` to denote that it is intended to be thread-local. Loci will flag the assignment of line 6 as a compile-time error because it breaks the thread-locality guarantee on `local`.

In cases where the thread needs to explicitly store thread-local variables on the heap, a `@Thread` field can be used. Class `NotSoLeaky` is a correct implementation of `Leaky` that uses the `ThreadLocal` API explicitly to store the contents of the `local` variable in a thread-local field. The same effect can be achieved in Loci by annotating a field as `@Thread`. This is done in class `Safe` where the field `safe` is thread-local. In this case, Loci will silently transform the code written by the programmer (at compile-time) into equivalent code which uses `ThreadLocal`.

To sum up, Loci guarantees that the contents of any field or variable annotated `@Thread` is and will remain thread-local. For variables this is done entirely at compile-time (by restricting assignments) and for fields the guarantee is obtained by translation into using the `ThreadLocal` API.

```
1  class Leaky extends Thread {        class Safe extends Thread {
2    String[] unsafe;                    @Shared String[] unsafe;
3                                        @Thread String[] safe;
4    public void run() {                 public void run() {
5      String[] local = ...;               @Thread String[] local = ...;
6      unsafe = local; // leak             unsafe = local; // wont compile
7                                          safe = local; // OK
8  } }                                 } }
9  ...                                 ...
10 Leaky t;                            Safe t;
11 ... = t.unsafe;                     ... = t.unsafe;
12                                     ... = t.safe; // no leak
```

```
1  class NotSoLeaky extends Thread {
2    ThreadLocal<String[]> safe = new ThreadLocal<String[]>();
3
4    public void run() { String[] local = ...; safe.set(local); }
5  }
6  ...
7  NotSoLeaky t = ...;
8  ... = t.safe.get();
```

Fig. 1. Enforcing thread-locality with Loci. Line 6 marked Leak in the leftmost example is prevented statically. On Line 12, the assignment from safe does not cause a leak, as different threads get different values when reading a thread-local field. Class NotSoLeaky is safe as it uses the the ThreadLocal API, Loci provides more convenient syntax for the same. Whenever ever a field is annotated @Thread, the Loci compiler will turn it into a ThreadLocal.

2.2 Logical View of the Heap

In Loci, the heap of a program with n threads is *logically* partitioned into n number of isolated heaps, called "heaplets", plus a shared heap. There is a one-to-one mapping between threads and heaplets. From now on, *heap* refers to the shared heap accessible by all threads, and *heaplets* refers to thread-local heaps. The Loci annotation system enforces the following simple properties on Java programs, shown in Fig. 2:

1. References from one heaplet into another are not allowed (⟶);
2. References from heaplets to the shared heap are unrestricted (- - ►);
3. References from the shared-heap into a heaplet must be stored in a thread-local field (•- - ►).

The third property above ensures that even though a reference into a heaplet ρ_i may exist on the shared heap, it is only accessible to the thread i to which ρ_i belongs. If another thread j reads the same field, it will either get a reference into its own heap ρ_j that it had written there before, or a thread-local copy of the

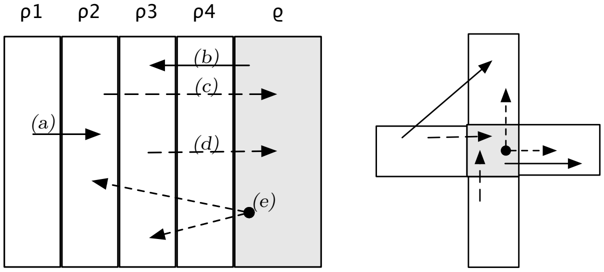

Fig. 2. Thread-Local Heaplets and a Shared Heap. The teal area (ϱ) is the shared heap, white areas ($\rho_1..\rho_4$) represent the thread-local heaplets. Solid arrows are invalid and correspond to property 1 in Sec. 2.2, dashed arrows are valid pointers into the shared heap (property 2), respectively 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.

default value of the field (which may be `null`). Effectively, there is a copy of each thread-local field for each active thread in the system and writes and reads of the same thread-local field by different threads access different copies. Together, these simple properties make heaplets effectively thread-local, and objects in the heaplets are thus safe from race conditions and data races.

2.3 Annotations

Loci uses two annotations, `@Thread` and `@Shared`, their semantics is summarized in Table 1. We distinguish between class-level annotation and annotations on types in declarations. Class-level annotations control how instances of a class can be used. Instances of classes annotated `@Shared` always live in the shared

Annotation	Level	
`@Shared`	Class	Instances are always allocated in the shared heap.
`@Thread`	Class	Instances can be allocated either in the shared heap or in heaplets.
–	Class	Equivalent to `@Shared`.
`@Shared`	Field	May point into the heap or into a heaplet.
`@Thread`	Field	Must refer to a heaplet-allocated value. Access to the object is through the `ThreadLocal` API.
–	Field	Treated as `@Shared` if the enclosing class is `@Shared`, and as a `@Thread` local variable otherwise. (`@Context` in the formalism.)
`@Shared`	Local	May point into the heap or into a heaplet.
`@Thread`	Local	Must refer to a heaplet-allocated value.
–	Local	Treated as `@Shared` if the enclosing class is `@Shared`, and `@Thread` otherwise. (Called `@Context` in the formalism.)

Table 1. Loci annotations.

```
1 @Thread class Foo { Foo f; }     @Shared class S { S f; }
2 @Shared Foo x;                   @Shared S x;
3 @Thread Foo y;                   @Thread S y; // Illegal
4 Foo z;                           S z = x = x.f;
5 @Shared Foo xx = x.f;
6 @Thread Foo yy = y.f;            @Thread class C { C m(S) { ... } }
7 Foo zz = z.f;                    @Thread C c;
8 yy.f = z; // Illegal             c = c.m(x.f);
```

Fig. 3. Loci by example. *Left:* `f` in `Foo` will live on same heap(let) as the enclosing obj. Thus, 5–7) are valid. Depending on the current context, `z` can be both shared and unshared. Thus, 8) is illegal. However, when we know the nature of the context, we can figure out the precise type (`y` in 6). *Right:* Line 3) demonstrates that `@Shared` classes cannot be pointed to by `@Thread` variables. Line 4) shows that an unannotated field in a `@Shared` class will also be shared. Line 6) trivially shows viewpoint adaptation.

heap. Instances of classes annotated `@Thread` may live on the shared heap, or on a thread-local heap. The class-level annotations control the implicit defaults used on types in the class. In a `@Shared` class, all types are implicitly shared. Unless explicitly declared `@Thread`, fields in a `@Shared` class point to objects in the shared heap. Fig. 3 contains some example uses of the annotations. In `@Thread`-annotated classes, the default annotation is empty (`@Context` in the formalism), which is a non-annotation only used implicitly. Empty is equivalent to `owner` in ownership types systems and means "the same as the enclosing instance." If `f` is an unannotated field in an instance `o` of a `@Thread` class, `f` will point to an object in the same heaplet as `o`, or into the shared heap iff `o` is in the shared heap. In practice, the implicit default value is right most of the time. Oftentimes, a single class-level annotation is all that is necessary. When accessing an unannotated field of a `@Thread` variable, the field's annotation will automatically default to `@Thread`. We refer to this as *viewpoint adaptation* and it is a simplified version of σ-substitution found in several ownership types formalisms [12, 13, 34, 26].

Rather than splitting classes into thread-local and not, a `@Thread` class can be used to instantiate both thread-local and shared objects. This makes code more flexible and reusable in both shared and unshared context which is important for library classes. The main restriction for `@Thread` classes is that they may not assign from **`this`** in a way that could invalidate isolation. This is simple to check statically—disallow values whose annotation is unknown to be stored in explicitly annotated fields or variables. Since globals are always `@Shared`, a `@Thread` class may not leak **`this`** into them and so the only way to invalidate thread-locality is by storing **`this`** in a constructor argument. To that end, if a constructor takes 0 arguments or all `@Thread` arguments, the new object will be thread-local. Otherwise it will be shared.

Fields, variables, parameters and return types annotated `@Thread` or `@Shared` point to thread-local respectively shared objects. If the annotation is empty, it is effectively the same as the current **`this`**. To maintain compatibility with existing

```
@Thread class RayTracerRunner extends RayTracer {
  @Shared Barrier br = null; # Barrier is shared between threads
  ...
}
...
// was thobjects[i] = new RayTracerRunner(i, wdt, hgt, brl);
thobjects[i] = new Thread() {
  public void run() {
    @Thread RayTracerRunner _ = new RayTracerRunner(i, wdt, hgt, brl);
    _.run(); // Start thread-local computation
  }
};
...
```

Fig. 4. Code from Raytracer in JavaGrande refactored with Loci annotations.

Java code, the default class-level annotation is `@Shared`. Thus, the class `Thread` is shared (which is the only sensible option), and as subclassing must preserve annotations in Loci (modulo for `Object`, see below), the derived thread classes in Fig. 1 must also be `@Shared`.

The root class `Object` is annotated `@Thread`. We treat it specially in that we allow it to be extended as `@Shared`. This is type safe, and as subtyping preserves annotations, cannot be used to confuse the type system.

Finally, Fig. 4 shows an example from Raytracer in the JavaGrande benchmarks where RayTracer classes all the classes that are used by RayTracer are annotated `@Thread`, modulo the shared barrier. The computation is thus entirely thread-local. The arrays of threads contain threads that simply start the thread-local computation.

2.4 Migrating Objects

Some concurrent programming idioms are characterized by phased access to objects. For example, in a producer-consumer pattern, an object is first accessed exclusively by the producer thread before being handed out to the consumer thread which then has sole access to the object. This goes beyond the kind of thread-locality expressible directly in Loci or most simple ownership type systems. Approaches based on linearity are feasible but they would overly complicate the type system. In Loci, when thread-local objects must migrate between threads it is necessary to perform a deep copy via the heap. Fig. 5 shows a method that copies an instance of `Foo` from Fig. 3 from a heaplet to the shared heap. Copying an object *directly* across heaplets is not yet supported, but we plan to investigate simple ways of doing this in the future.

For now, two threads in a producer-consumer relationship wishing to transfer unshared objects across via for example a shared queue, must copy the objects twice and place them on the shared heap. As noted in Sec. 2.3, most newly created instances of `@Thread` classes can safely be stored anywhere.

Two minor technicalities prevent us from doing direct inter-heaplet migration. Firstly, we cannot name a field belonging to another thread. Second, the annotations can only express references into the current thread's heaplet or the shared heap. Extending the system to allow both is relatively straightforward. Writing to another thread's field can be as simple as extending `ThreadLocal` with a `put(Thread t, Object val)` method. A `@StackLocal` annotation could also be employed to type the `put()` method, and at the same time obviate the need for two deep copy methods, but would further complicate the system, especially when preserving aliasing through a cache like in Fig. 5.

```
@Shared Foo copyToShared(Cache cache) {
  if (cache.hasKey(this)) return cache.get(this);
  @Shared Foo copy = new Foo();
  cache.put(this, copy);
  if (f != null) {
    copy.f = f.copyToShared(cache);
  }
  return copy;
}
```

Fig. 5. A simple deep copy method for `Foo` that makes a shared copy of a possibly thread-local object. Cache is a `@Thread` map from keys to `@Shared` objects. Removing `@Shared` would make the method return a thread-local copy, as `Foo` is a `@Thread` class.

2.5 Run-Time Overhead

Loci does not add run-time overhead over the equivalent Java programs. It does not need to store to what heap(let) an object belongs at run-time. `@Thread`-annotated local variables, parameters and method returns all live on the stack and are thus effectively thread-local without the need for any additional magic. Most notably, implicitly thread-local fields (e.g., `f` in Fig. 3) in `@Thread`-annotated classes do not incur any additional overhead. The key realization is that access to the enclosing object acts as a guard (similar observations have been done elsewhere, e.g., [13, 34, 16, 18]):

> *If the enclosing object is only reachable by its "owning thread", the same holds for all objects pointed to by its non-`@Shared` fields.*

The overhead of `@Thread`-annotated *fields* is due to their implementation using Java's `ThreadLocal` API. Micro benchmarks suggest that access to a thread-local field is about 8 times slower than a regular field access. In our experience, fields only need to be annotated `@Thread` in the places where the Java program would have used the `ThreadLocal` API explicitly.

3 A Formal Account of Loci

We formalize our system in a subset of Java. For brevity, we omit commonly omitted features, such as overriding, interfaces, exceptions, final variables, primitive data types, arrays and generics. Generics is not yet supported by Loci as Java does not yet support annotations on type parameters. Should JSR 308, "Type Annotations" be accepted, adding support for annotating type parameters would be straightforward and would improve our story for collection classes.

3.1 Syntax and Static Semantics

Loci's syntax is shown in Fig. 6. For simplicity, we use an explicit annotation `@Context`, instead of implicit annotations. Without loss of generality, we use a "named form," where the results of field and variable accesses, method calls and instantiations must be immediately stored in a variable or field. For simplicity, all rules have an implicit program P in which classes are looked up. We use the right-associative viewpoint-adaptation operator $\oplus$ to expand the default `@Context` annotation thus:

$$\alpha_1 \oplus \alpha_2\ c = \begin{cases} \alpha_1\ c \text{ if } \alpha_2 = \texttt{@Context} \\ \alpha_2\ c \text{ otherwise} \end{cases} \qquad \alpha_1 \oplus \alpha_2 \oplus \alpha_3\ c = \alpha_1 \oplus (\alpha_2 \oplus \alpha_3\ c)$$

For brevity, we assume that $\mathsf{fields}(c) = \overline{\tau}\ \overline{f}$ where $\overline{f}$ are all fields in c and super classes of c. We use the shorthand $\mathsf{fields}(c.f) = \tau$ to say that field f in class c has the type τ. For methods, we assume the existence of $\mathsf{mtype}(c.m) = \overline{\tau} \to \tau$ and $\mathsf{mbody}(c.m) = (\overline{x}, s;\texttt{return}\ x)$ where $\tau\ m(\overline{\tau}\ \overline{x})\{\ s;\texttt{return}\ x\ \}$ is declared in the most direct superclass to c that declares m. We sometimes write (SUB-$*$) to denote all rules starting with "SUB" and ($*$-VAR) for all rules ending with "VAR."

We say that a program P is well-formed if all class definitions are well formed. By construction, all class hierarchies are rooted in `Object`. For simplicity, `Object`

$$\begin{array}{rcll}
P & ::= & \overline{cd} & \textit{program} \\
cd & ::= & \alpha\ \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{fd}\ \overline{md}\ \} & \textit{class declaration} \\
fd & ::= & \tau\, f & \textit{field} \\
md & ::= & \tau\, m(\overline{\tau}\ \overline{x})\ \{\ s;\texttt{return}\ y\ \} & \textit{method} \\
s & ::= & s;s \mid \texttt{skip} \mid x = y.f \mid x = y \mid y.f = z \mid \tau\ x \mid & \textit{statement} \\
 & & x = \texttt{new}\ \tau() \mid x = y.m(\overline{z}) \mid x = \texttt{start}\ c() & \\
\tau & ::= & \alpha\ c & \textit{type} \\
\alpha & ::= & \texttt{@Thread} \mid \texttt{@Shared} \mid \texttt{@Context} & \textit{annotations} \\
 & & & \\
E & ::= & [] \mid E[x : \tau] & \textit{local type environment}
\end{array}$$

Fig. 6. Loci's syntax. c, d are class names, f, m are field and method names, and x, y, z are names of variables or parameters respectively, where $x \neq \texttt{this}$. For simplicity, we assume that names of classes, fields, methods and variables are unique. The special variable **ret** and **return** only appears in the dynamic syntax and semantics.

is an empty class with no superclasses that is annotated `@Thread`. A user-defined class is well-formed if it abides by (T-CLASS).

$$\textsc{(t-class)}\quad \dfrac{\begin{array}{c}\mathsf{fields}(d) = fd_2 \quad \mathsf{methods}(d) = md_2 \quad \mathsf{annote}(d) = \alpha_2 \\ \forall m \in \mathsf{names}(\overline{md_1}) \cap \mathsf{names}(\overline{md_2}).\ \mathsf{mtype}(c.m) = \mathsf{mtype}(d.m) \\ \mathsf{names}(\overline{fd_1}) \cap \mathsf{names}(\overline{fd_2}) = \emptyset \\ \alpha_1 \neq \texttt{@Context} \qquad (\alpha_1 = \alpha_2 \vee d = \texttt{Object}) \qquad \alpha_1 \vdash \overline{fd_1} \qquad \alpha_1\ c \vdash \overline{md_1}\end{array}}{\vdash \alpha_1\ \textbf{class}\ c\ \textbf{extends}\ d\ \texttt{\{}\ \overline{fd_1}\ \overline{md_1}\ \texttt{\}}}$$

Notably, subclassing and overriding must preserve annotations, overriding is not supported, and `@Context` is not a valid class-level annotation.

$$\textsc{(t-field)}\quad \dfrac{\vdash \alpha_1 \oplus \alpha_2\ c}{\alpha_1 \vdash \alpha_2\ c\ f} \qquad \textsc{(t-method)}\quad \dfrac{\begin{array}{c}(\alpha_1 = \texttt{@Thread} \wedge \alpha_2 = \texttt{@Context}) \ \vee\ \alpha_2 = \texttt{@Shared} \\ \textbf{this} : \alpha_2\ c, \overline{x} : \overline{\tau} \vdash s; E \qquad E(y) \leq \alpha_2 \oplus \tau\end{array}}{\alpha_1\ c \vdash \tau\ m(\overline{\tau}\ \overline{x})\texttt{\{}\ s\texttt{;}\textbf{return}\ y\ \texttt{\}}}$$

(T-FIELD) uses the viewpoint-adaptation operator $\oplus$ on annotations and types. α_2 is the annotation on the field and α_1 is the annotation of the declaring class used if α_2 is `@Context`. The class of the field must be valid with respect to the resulting annotation. This is similar to σ-substitution found in ownership types type systems and is used frequently in the formalism.

In (T-METHOD), the type of **this** depends on the enclosing class. In a `@Thread` class, **this** is `@Context` and otherwise `@Shared`. This is because `@Thread` classes can be used to create both shared and thread-local instances.

Statements. The statements should be straightforward to follow for anyone familiar with Java. Remember, $x \neq \textbf{this}$.

$$\textsc{(t-sequence)}\quad \dfrac{E \vdash s_1; E_1 \quad E_1 \vdash s_2; E_2}{E \vdash s_1\texttt{;}s_2; E_2} \qquad \textsc{(t-skip)}\quad \dfrac{}{E \vdash \textbf{skip}; E} \qquad \textsc{(t-assign)}\quad \dfrac{E(y) \leq E(x)}{E \vdash x = y; E}$$

$$\textsc{(t-select)}\quad \dfrac{\begin{array}{c}E(y) = \alpha\ c \\ \mathsf{fields}(c.f) = \tau' \\ \alpha \oplus \tau' \leq E(x)\end{array}}{E \vdash x = y.f; E} \qquad \textsc{(t-update)}\quad \dfrac{\begin{array}{c}E(y) = \alpha\ c \\ \mathsf{fields}(c.f) = \tau \\ E(z) \leq \alpha \oplus \tau\end{array}}{E \vdash y.f = z; E} \qquad \textsc{(t-decl)}\quad \dfrac{\begin{array}{c}x \notin dom(E) \\ E(\textbf{this}) = \alpha\ c \\ E' = E[x : \alpha \oplus \tau]\end{array}}{E \vdash \tau\ x; E'}$$

(T-SELECT) and (T-UPDATE) applies $\oplus$ to the annotation on the target and the field to possibly expand `@Context`s. Note that (T-DECL) replaces `@Context` with the annotation of the current **this** (which may be `@Context`).

$$\textsc{(t-new)}\quad \dfrac{\vdash \tau \qquad \tau \leq E(x)}{E \vdash x = \textbf{new}\ \tau\texttt{()}; E} \qquad \textsc{(t-call)}\quad \dfrac{\begin{array}{c}E(y) = \alpha\ c \quad \mathsf{mtype}(c.m) = \overline{\tau} \rightarrow \tau' \\ E(\overline{z}) \leq \alpha \oplus \overline{\tau} \quad \alpha \oplus \tau' \leq E(x)\end{array}}{E \vdash x = y.m\texttt{(}\overline{z}\texttt{)}; E} \qquad \textsc{(t-fork)}\quad \dfrac{\begin{array}{c}\mathsf{mtype}(c.\texttt{run}) = \epsilon \rightarrow \tau \\ E(x) = \texttt{@Shared}\ c\end{array}}{E \vdash x = \textbf{start}\ c\texttt{()}; E}$$

Similar to how Java deals with threads, the **`start`** operation only works on classes that have a 0-arity `run` method (denoted by ϵ parameter types).

Subtyping, Types. The subtyping relation $\leq$ is the transitive relation closed under the rules below. $\mathsf{annote}(c)$ returns the annotation on the class c or `@Thread` if $c = \texttt{Object}$.

$$\overset{(\textsc{sub-direct})}{\frac{\alpha\ \texttt{class}\ c\ \texttt{extends}\ d\ \cdots}{c \leq d}} \qquad \overset{(\textsc{sub-trans})}{\frac{c \leq c' \qquad c' \leq d}{c \leq d}} \qquad \overset{(\textsc{sub-self})}{\frac{}{c \leq c}}$$

$$\overset{(\textsc{sub-annote})}{\frac{\vdash \alpha\ c \qquad \vdash \alpha\ d \qquad c \leq d}{\alpha\, c \leq \alpha\, d}} \qquad \overset{(\textsc{type})}{\frac{\mathsf{annote}(c) = \texttt{@Shared} \Rightarrow \alpha = \texttt{@Shared}}{\vdash \alpha\ c}}$$

By (sub-annote), subtyping must preserve annotations. Most importantly, though, `Object` may be subclassed as both `@Shared` and `@Thread`.

3.2 Dynamic Semantics

We formulate Loci's dynamic semantics as a small-step operational semantics. See Fig. 7 for syntax. A Loci configuration $H;\overline{T}$ consists of a single heap H of locations mapped to objects tagged to denote to what heap(let) they belong to and a collection of threads. Each thread T has its own stack, plus a thread id denoted ρ. An object belonging to the thread ρ will be tagged ρ in its second compartment. We use ϱ in the syntactic category ρ to denote the shared heap. Thread-scheduling is modeled as a non-deterministic choice in (d-schedule). A configuration with a thread scheduled to run is denoted $H;T;\overline{T}$. For convenience, we write $H(\iota.f)$ as a shorthand for $H(\iota)(f)$ and $H(\iota.f) := v$ for $H[\iota \mapsto o[f \mapsto v]]$. We denote the look-up of a non-existent field $H(\iota.f) = \bot$ (where $\bot \neq v$), which can happen due to lazy creation of thread-local fields. The initial configuration has the form $[]; (\langle [], s;\texttt{return}\ x\rangle, \rho)$, i.e., there is only one thread on start-up, and the initial stack frame and heap are empty. The relation $(\rightarrow)$ is the reduction step on configurations.

$$\begin{array}{lr|lr}
H ::= [] \mid H[\iota \mapsto o] & \textit{heap} & F ::= [] \mid F[y \mapsto v] & \textit{stack frame} \\
T ::= (S, \rho) \mid (\texttt{NPE}, \rho) & \textit{thread} & o ::= c(\rho, F) & \textit{object} \\
S ::= \epsilon \mid S\,\langle F, s\rangle & \textit{stack} & v ::= \iota \mid \texttt{null} & \textit{value}
\end{array}$$

Fig. 7. Syntax for heaps, threads, stacks, frame, objects and values. For brevity, we unify stack frame and object fields. To distinguish, we use f for fields and y for variables.

The rule (d-schedule) non-deterministically picks one thread for execution. The rules (d-finish) and (d-dead) remove threads that are fully reduced from the system. `NPE` is a thread that's dead from a null-pointer error.

$$\overset{\textsc{(d-schedule)}}{\frac{H;\overline{T}\,\overline{T'}\,T \to H';\overline{T''}}{H;\overline{T}\,T\,\overline{T'} \to H';\overline{T''}}} \qquad \overset{\textsc{(d-finished)}}{\frac{}{H;\overline{T}\,(\langle F, \textbf{return}\ x\rangle, \rho) \to H;\overline{T}}} \qquad \overset{\textsc{(d-dead)}}{\frac{}{H;\overline{T}\,(\texttt{NPE}, \rho) \to H;\overline{T}}}$$

Local variable declaration and assignment offer no surprises.

$$\overset{\textsc{(d-assign)}}{\frac{F(y) = v \qquad T = (S\,\langle F[x = v], s\rangle, \rho)}{H;\overline{T}\,(S\,\langle F, x = y; s\rangle, \rho) \to H;\overline{T}\,T}} \qquad \overset{\textsc{(d-decl)}}{\frac{T = (S\,\langle F[x \mapsto \texttt{null}], s\rangle, \rho)}{H;\overline{T}\,(S\,\langle F, \tau\ x; s\rangle, \rho) \to H;\overline{T}\,T}}$$

$$\overset{\textsc{(d-skip)}}{\frac{}{H;\overline{T}\,(S\,\langle F, \textbf{skip}; s\rangle, \rho) \to H;\overline{T}\,(S\,\langle F, s\rangle, \rho)}}$$

We model thread-local variables as zero or more variables indexed by the thread id ρ—a thread ρ accessing a `@Thread` field `f` returns the contents of the field $\texttt{f}_\rho$.

$$\mathsf{sel}(\iota, c.f, H, \rho) = \begin{cases} \texttt{null} & \text{if } \mathsf{fields}(c.f) = \texttt{@Thread}\ d \wedge H(\iota.f_\rho) = \bot \\ v & \text{if } \mathsf{fields}(c.f) = \texttt{@Thread}\ d \wedge H(\iota.f_\rho) = v \\ v' & \text{if } H(\iota.f) = v' \end{cases}$$

The predicate $\mathsf{sel}()$ returns the value of the request field, or, if the field is thread-local, the value of the field indexed by the current thread.

$$\overset{\textsc{(d-select)}}{\frac{F(y) = \iota \quad H(\iota) = c(\cdots) \quad \mathsf{sel}(\iota, c.f, H, \rho) = v \quad T' = (S\,\langle F[x = v], s\rangle, \rho)}{H;\overline{T}\,(S\,\langle F, x = y.f; s\rangle, \rho) \to H;\overline{T}\,T'}}$$

Missing thread-local fields are given the value `null`. An alternative would be to create a copy for every thread in the system, but the above solution felt somewhat closer to the semantics of the `ThreadLocal` API, which calls `initialValue()` on the first read of a field by a particular thread. Like reading, writing a `@Thread` field updates the copy of the field indexed by the current thread's id.

$$\mathsf{upd}(\iota, c.f, H, \rho, v) = \begin{cases} H(\iota.f_\rho) := v & \text{if } \mathsf{fields}(c.f) = \texttt{@Thread}\ c \\ H(\iota.f) := v & \text{otherwise} \end{cases}$$

$$\overset{\textsc{(d-update)}}{\frac{F(y) = \iota \quad H(\iota) = c(\cdots) \quad H' = \mathsf{upd}(\iota, c.f, H, \rho, F(z))}{H;\overline{T}\,(S\,\langle F, y.f = z; s\rangle, \rho) \to H';\overline{T}\,(S\,\langle F, s\rangle, \rho)}}$$

We have omitted constructors (see Sec. 4.1 for a discussion on how to deal with them). Thus, a new instance is always thread-local and can subsequently be placed either on the shared heap or in the current heaplet. This is decided by the annotation of the target variable for the instantiation.

$$\mathsf{reg}(\alpha, \rho, \rho_1) = \begin{cases} \rho & \text{if } \alpha = \texttt{@Thread} \\ \rho_1 & \text{if } \alpha = \texttt{@Context} \\ \varrho & \text{if } \alpha = \texttt{@Shared} \end{cases}$$

The predicate $\mathsf{reg}()$ “registers” a newly created instance with a certain thread.

$$\overset{\textsc{(d-new)}}{\frac{\begin{array}{c} H(F(\textbf{this})) = d(\rho_1, F_1) \quad \iota \text{ is fresh} \quad \mathsf{names}(\mathsf{fields}(c)) = \overline{f} \\ H' = H[\iota \mapsto c(\mathsf{reg}(\alpha, \rho, \rho_1), \overline{f} \mapsto \overline{\texttt{null}})] \end{array}}{H; \overline{T}\,(S\,\langle F, x = \textbf{new}\ \alpha\ c\texttt{()}; s\rangle, \rho) \to H'; \overline{T}\,(S\,\langle F[x \mapsto \iota], s\rangle, \rho)}}$$

If the target variable is `@Context`-annotated, the class is stored in the same heap or heaplet as the current **this**. We use the special variable **ret** to capture return values. The only assignment to **ret** is through a return which assigns the **ret** of the underlying stack frame.

$$\overset{\textsc{(d-return)}}{\frac{F(y) = v \qquad T = (S\,\langle F'[\textbf{ret} \mapsto v], s'\rangle, \rho)}{H; \overline{T}\,(S\,\langle F', s'\rangle\langle F, \textbf{return}\ y\rangle, \rho) \to H; \overline{T}\,T}}$$

$$\overset{\textsc{(d-call)}}{\frac{\begin{array}{c} F(y) = \iota \quad F(\overline{z}) = \overline{v} \quad H(\iota) = c(\cdots) \quad \mathsf{mbody}(c.m) = (\overline{x'}, s'\,\textbf{;return}\ y') \\ F' = \textbf{this} \mapsto \iota, \overline{x'} \mapsto \overline{v} \quad S' = S\,\langle F, x = \textbf{ret}; s\rangle \end{array}}{H; \overline{T}\,(S\,\langle F, x = y.m\texttt{(}\overline{z}\texttt{)}; s\rangle, \rho) \to H; \overline{T}\,(S'\,\langle F', s'\,\textbf{;return}\ y'\rangle, \rho)}}$$

An invocation $x = y.m\texttt{()}$ is rewritten into $x = \textbf{ret}$ and the method’s body is executed on a new stack frame eventually assigning **ret** as the result of a return.

$$\overset{\textsc{(d-fork)}}{\frac{\begin{array}{c} \iota, \rho' \text{ are fresh} \quad \mathsf{names}(\mathsf{fields}(c)) = \overline{f} \quad H' = H[\iota \mapsto c(\varrho, \overline{f} \mapsto \overline{\texttt{null}})] \\ \mathsf{mbody}(c.\mathsf{run}) = (\epsilon, s'\,\textbf{;return}\ y') \quad T = (\langle[\textbf{this} \mapsto \iota], s'\,\textbf{;return}\ y'\rangle, \rho') \end{array}}{H; \overline{T}\,(S\,\langle F, x = \textbf{start}\ c\texttt{()}; s\rangle, \rho) \to H'; \overline{T}\,T\,(S\,\langle F[x \mapsto \iota], s\rangle, \rho)}}$$

The (D-FORK) operations adds a thread to the system and is a simplified union of Java’s `new` and `start`. The new thread object is created on the shared area, forcing its thread-local data to be stored either on the stack of the `run` method, or in a thread-local field. Adding thread-local threads to the system would be as simple as introducing a **start** operation that returns `null`.

For brevity, null-pointer exceptions kill the entire thread rather than propagate an error through the execution. The semantics is effectively the same.

$$\overset{\textsc{(d-select-npe)}}{\frac{}{H; \overline{T}\,(S\,\langle F[y \mapsto \texttt{null}], x = y.f; s\rangle, \rho) \to H; \overline{T}\,(\texttt{NPE}, \rho)}}$$

$$\frac{}{H; \overline{T}\,(S\,\langle F[y \mapsto \mathtt{null}], y.f = z; s\rangle, \rho) \rightarrow H; \overline{T}\,(\mathtt{NPE}, \rho)}\;\textsc{(d-update-npe)}$$

$$\frac{}{H; \overline{T}\,(S\,\langle F[y \mapsto \mathtt{null}], x = y.m(\overline{z}); s\rangle, \rho) \rightarrow H; \overline{T}\,(\mathtt{NPE}, \rho)}\;\textsc{(d-call-npe)}$$

3.3 Meta-Theory

Well-Formedness Rules We now present the rules for well-formed configurations, heaps and stacks. $\mathit{\Gamma}$ is the store-type and has the syntax $\mathit{\Gamma} ::= \epsilon \mid \mathit{\Gamma}[\iota : \tau]$.

$$\mathsf{tid}(H, \iota) = \begin{cases} \rho & \text{if } H(\iota) = c(\rho, F) \\ \bot & \text{otherwise} \end{cases} \qquad \mathsf{tid}(T) = \begin{cases} \rho & \text{if } T = (S, \rho) \\ \rho & \text{if } T = (\mathtt{NPE}, \rho) \end{cases}$$

In the following rules, we make use of the auxiliary function $\mathsf{tid}(T)$ that extracts a thread's id, and $\mathsf{tid}(H, \iota)$, that looks up the thread id of an object on the heap. (wf-config) states that a configuration is well-formed if there is a well-formed store typing $\mathit{\Gamma}$ which type the heap H and if all threads have distinct ids and are well-formed.

$$\frac{\vdash \mathit{\Gamma} \qquad \mathit{\Gamma}; H \vdash H \qquad \mathsf{tid}(\overline{T}) \text{ distinct} \qquad \forall\,(S, \rho) \in \overline{T}\;.\;\mathit{\Gamma}; H \vdash_\rho S}{\mathit{\Gamma} \vdash H; \overline{T}}\;\textsc{(wf-config)}$$

$$\frac{}{\vdash []}\;\textsc{(wf-}\mathit{\Gamma}\textsc{-0)} \qquad \frac{\vdash \mathit{\Gamma} \quad \vdash \alpha\ c \qquad \alpha \neq \mathtt{@Context}}{\vdash \mathit{\Gamma}[\iota : \alpha\ c]}\;\textsc{(wf-}\mathit{\Gamma}\textsc{-1)} \qquad \frac{}{\mathit{\Gamma}; H \vdash_\rho []}\;\textsc{(wf-thread-0)} \qquad \frac{\mathit{\Gamma}; H \vdash_\rho S \qquad \mathit{\Gamma}; H; E \vdash_\rho F \quad E \vdash s; E'}{\mathit{\Gamma}; H \vdash_\rho S\,\langle F, s\rangle}\;\textsc{(wf-thread-1)}$$

An object is well-formed if all its fields point to locations on the heap. Thread-local objects must have the same id as the current thread or, otherwise, the id of the shared heap. Note that `@Context` does not appear on types in $\mathit{\Gamma}$.

$$\frac{}{\mathit{\Gamma}; H' \vdash_\rho []}\;\textsc{(wf-heap-}\epsilon\textsc{)} \qquad \frac{\mathit{\Gamma}; H' \vdash_\rho H \quad \mathit{\Gamma}(\iota) = \mathtt{@Shared}\ c \qquad \mathsf{fields}(c) = E \qquad \mathit{\Gamma}; H'; E \vdash_\varrho F}{\mathit{\Gamma}; H' \vdash_\rho H[\iota \mapsto c(\varrho, F)]}\;\textsc{(wf-heap-shared)}$$

$$\frac{\mathit{\Gamma}; H' \vdash_\rho H \quad \mathit{\Gamma}(\iota) = \mathtt{@Thread}\ c \qquad \mathsf{fields}(c) = E \qquad \mathit{\Gamma}; H'; E \vdash_\rho F}{\mathit{\Gamma}; H' \vdash_\rho H[\iota \mapsto c(\rho, F)]}\;\textsc{(wf-heap-thread)}$$

Due to the treatment of thread-local fields, rules for well-formed fields are a bit more complex that usual for a Java-like language. (wf-field-ϵ) captures that thread-local fields may not yet have been initialized.

$$\frac{}{\mathit{\Gamma}; H; E \vdash_\rho []}\;\textsc{(wf-field-}\epsilon\textsc{)} \qquad \frac{\mathit{\Gamma}; H; E \vdash_\rho F \qquad E(f) = \tau}{\mathit{\Gamma}; H; E \vdash_\rho F[f \mapsto \mathtt{null}]}\;\textsc{(wf-field-null)} \qquad \frac{\mathit{\Gamma}; H; E \vdash_\rho F \qquad E(f) = \tau}{\mathit{\Gamma}; H; E \vdash_\rho F[f_{\rho'} \mapsto \mathtt{null}]}\;\textsc{(wf-field-thread-null)}$$

$$\frac{\Gamma; H; E \vdash_\rho F \qquad E(f) = \tau \qquad \mathsf{tid}(H, \iota) = \rho' \qquad \Gamma(\iota) \leq \tau}{\Gamma; H; E \vdash_\rho F[f_{\rho'} \mapsto \iota]}\ (\textsc{wf-field-thread})$$

$$\frac{\begin{array}{c}\Gamma; H; E \vdash_\varrho F \quad E(f) = \texttt{@Shared}\ c \\ \Gamma(\iota) \leq \texttt{@Shared}\ c \qquad \mathsf{tid}(H, \iota) = \varrho\end{array}}{\Gamma; H; E \vdash_\rho F[f \mapsto \iota]}\ (\textsc{wf-field-shared}) \qquad \frac{\begin{array}{c}\Gamma; H; E \vdash_\rho F \quad E(f) = \texttt{@Context}\ c \\ \Gamma(\iota) \leq \alpha\ c \quad \mathsf{tid}(H, \iota) = \rho\end{array}}{\Gamma; H; E \vdash_\rho F[f \mapsto \iota]}\ (\textsc{wf-field-context})$$

Notably, `@Shared` fields point to objects on the shared heap, `@Context` fields point to objects on the same heap(let) as the current **`this`**, and `@Thread` fields have ≥ 0 copies subscripted with the same thread id as the object they point to.

For stack frames, a pointer in a `@Shared` field points to an object on the shared heap and a pointer in a non-shared field points to an object on the same heap(let) as the current **`this`**.

$$\frac{}{\Gamma; H \vdash_\rho [], []}\ (\textsc{wf-frame-0}) \qquad \frac{\begin{array}{c}\Gamma; H \vdash_\rho E, F \\ \mathsf{tid}(H, \iota) = \rho' \quad \Gamma(\iota) \leq \alpha\ c \\ \alpha = \texttt{@Shared} \Rightarrow \rho' = \varrho \\ \alpha \neq \texttt{@Shared} \Rightarrow \rho' = \rho\end{array}}{\Gamma; H \vdash_\rho E[y : \alpha\ c], F[y \mapsto \iota]}\ (\textsc{wf-frame-1}) \qquad \frac{\Gamma; H \vdash_\rho E, F}{\Gamma; H \vdash_\rho E[y : \tau], F[y \mapsto \texttt{null}]}\ (\textsc{wf-frame-2})$$

Invariants Informally, Loci enforces the following property:

$$\boxed{\textit{A thread } \rho \textit{ can only access objects in heaplet } \rho \textit{ or on the shared heap } \varrho.}$$

We formulate this in two theorems, the first of which says that pointers in variables on a stack frame in a thread ρ either point to objects in ρ or in ϱ, and the second that evaluating a field access in thread ρ results in a pointer to either an object in ρ or in ϱ (or is a `null`-pointer).

Theorem 1. *Local variables point into shared heap or current heaplet. If* $\Gamma; E \vdash H; \overline{T}\ (S\ \langle F, s\rangle, \rho)$*, then* $\forall \iota \in rng(F).\ \mathsf{tid}(H, \iota) \in \{\varrho, \rho\}$*.*

Proof. Follows by straightforward induction on s. (wf-frame-1) and (wf-frame-2) are key. □

Theorem 2. *Field accesses yield pointers to shared heap or current heaplet. Let s be a field access $x = y.f$. If* $\Gamma; E \vdash H; \overline{T}\ (S\ \langle F, s\rangle, \rho)$*,* $H; \overline{T}\ (S\ \langle F, s\rangle, \rho) \rightarrow H'; \overline{T'}\ (S'\ \langle F', s'\rangle, \rho)$*, and* $F'(x) = \iota$*, then* $\mathsf{tid}(H', \iota) \in \{\varrho, \rho\}$*.*

Proof. The proof is by derivation on $\Gamma; E \vdash H; \overline{T}\ (S\ \langle F, x = y.f\rangle, \rho)$ relying on the fact that ρ is threaded through a computation and that `@Context`-annotated fields point to the heaplet of its enclosing object. By the rules for well-formed configurations, heaps and fields, $H(F(y).f) = \iota$ and $\mathsf{fields}(c.f) = \alpha\ c'$ implies $\mathsf{tid}(H, \iota) = \rho'$ s.t. (a) $\alpha =$ `@Shared` implies $\rho' = \varrho$, (b) $\alpha =$ `@Context` implies $\rho' = \rho''$ s.t. $\mathsf{tid}(H(F(y))) = \rho''$, and (c) $\alpha =$ `@Thread` implies $f = f_\rho \wedge \rho' = \rho$. Cases (a) and (c) immediately satisfy the theorem and case (b) follows immediately from Theorem 1 that gives $\rho'' \in \{\rho, \varrho\}$. □

Type Soundness We prove type soundness in the standard fashion of progress plus preservation [33]. In this context, preservation means that reduction does not invalidate the store typing.

Theorem 3. *Preservation. If* $\Gamma \vdash H; \overline{T}$*, and* $H; \overline{T} \rightarrow H'; \overline{T'}$*, then there exists a* Γ' *s.t.* $\Gamma' \vdash H'; \overline{T'}$*.*

Proof. The proof is straightforward by structural induction. There are no surprising cases. □

Theorem 4. *Progress. If* $\Gamma \vdash H; \overline{T}$*, then there exists a reduction such that* $H; \overline{T} \rightarrow H'; \overline{T'}$*.*

Proof. The proof is straightforward by structural induction on the shape of T where most cases are immediate. The slightly more intricate cases, (T-SELECT), (T-UPDATE) and (T-CALL) are all guarded by ($*$-NPE) versions of the rule that deal with null-dereferencing. By (WF-HEAP-$*$) and (WF-FIELD-$*$), a well-formed object $c(\rho, F)$ has all non-`@Thread` fields in F. By (SELECT-FIRST-THREAD), accessing an “undefined” thread-local field does not get stuck. Last, the only ways in which (D-NEW) or (D-FORK) could get stuck is if we cannot produce fresh ι’s or ρ’s, which is not modeled by our system. □

4 Loci for Eclipse

We have implemented Loci as an Eclipse plug-in. The plug-in supports most of Java, modulo generics, checking of native code, and reflection. The tool implements static checking of `@Shared`, `@Thread`, and the implicit `@Context` annotation. Currently, the tool gives a warning rather than an error when it detects a violation. Fig. 8 shows how `@Thread` fields are desugared into uses of the `ThreadLocal` API before compilation. The tool ignores primitive types and immutables, like strings and boxed primitives. To minimize the annotation burden, in a `@Thread` class, `@Context` is the implicit default annotation for a type of `@Thread` class, and `@Shared` for a type of `@Shared` class. In a `@Shared` class, the implicit annotation is `@Shared`. Notably, there is no keyword for `@Context`, it is only used implicitly. While it would have been more reasonable to default unannotated classes to `@Thread`, this would have had the drawback of requiring invasive changes to legacy code. To give existing Java programs a valid Loci interpretation, unannotated classes are implicitly `@Shared`, with the exception of `Object`.

```
@Thread class Foo {
  @Thread Foo foo = null;





  Object x;
  void bar() {
    @Thread Foo f = foo;
    foo = f;
  }
}
```

```
@Thread class Foo {
  @Thread ThreadLocal<Foo> foo =
    new ThreadLocal<Foo>() {
      Foo initialValue() {
        return null;
      }
    };
  Object x;
  void bar() {
    @Thread Foo f = foo.get();
    foo.set(f);
  }
}
```

Fig. 8. Sugared view and corresponding desugared view of a `@Thread` class. "Ensugar" and "desugar" buttons in the tool allows the user to switch back and forth between these views. Notably, `@Thread`-annotated local variables (`f` above) do not need to be implemented using the `ThreadLocal` API.

4.1 Extending Loci to Full Java

In this section, we address some of the interplay with Java not visible from the formalization, due to simplifications or Java conventions.

Anonymous classes. Loci fully supports anonymous classes. Anonymous classes automatically inherit the annotation of the superclass or interface. In the case of an interface without a class-level annotation, Loci currently requires the resulting instance to be stored immediately in a local variable and infers the annotation from the variable's type. Ambiguous instantiations of this kind could be solved by annotating the instantiated type.

Arrays. Loci supports three kinds of arrays (of any dimension):

1. Thread-local arrays of pointers to shared objects or primitives
2. Thread-local arrays of pointers to thread-local objects
3. Shared arrays of pointers to shared objects or primitives

If `Foo` and `Bar` are a `@Shared` respectively a `@Thread` class, then `@Thread Foo[]` is an array of the first kind, `@Thread Bar[]` the second, and `@Shared Foo[]` and `@Shared Bar[]` are of the third kind. The reason why `@Shared Bar[]` is a shared array of shared objects rather than an shared array of thread-local objects is because this case can be easily modeled by `@Thread Bar[]`. This frees `@Shared Bar[]` up to allow using `Bar` as a shared class in `@Shared` arrays. When using array initializers, Loci will inspect the annotations on the values used to initialize the array to infer whether the compartments of the array should be `@Thread` or `@Shared`, similar to constructor arguments in instantiation.

Interfaces. Unless explicitly annotated, we treat Java interfaces as implicitly annotated `@Context`, even on the class level (this is supported in the tool but omitted from the formalism since it does not deal with interfaces). When implemented, we apply the $\oplus$ operator using the annotation of the implementing class to get the annotation to which the implementing class must correspond. This allows us to reuse interfaces across different classes with different thread-local behavior.

Constructors and Instantiation. As `@Shared` classes are semantically equivalent to Java classes, their constructors require no special treatment. For `@Thread` classes, the story is different. Modulo constructor arguments, it is easy to see that a `@Thread` class instance constitutes a "free" value. The type of **`this`** is `@Context SomeClass` and as the types of static fields and methods must use either `@Shared` or `@Thread`, the class is effectively prevented from leaking itself. If a class "uses" `@Context` to annotate types of parameters to its constructor, we can derive the annotation of the new instance by looking at how these parameters are instantiated. If a **`new`** is only valid if the instance was shared (i.e., it binds a `@Shared` argument to a `@Context` parameter), the new object lives in the shared heap and the result type of the instantiation is `@Shared`. If `@Context` parameters are bound only to `@Context`, the resulting type is `@Context`. If both bindings are required, the instantiation is invalid as the object's type would have to be both `@Shared` and `@Thread`.

Inner and Nested classes. `@Shared` classes can have nested `@Thread` classes and vice versa. The inverse poses a problem, though, as the nested class instance has access to the enclosing instance, which could be thread-local. For this reason, Loci only allows instantiating `@Shared` inner classes inside non-thread-local instance. For example, given

```
@Thread class Foo { @Shared class Bar { } }
@Thread Foo f1;
@Shared Foo f2;
Foo f3;
```

we are allowed to do `f2.new Bar()` but not `f1.new Bar()` as the resulting shared instance would break thread-locality of the object in `f1`. Naturally, `f3.new Bar()` is only allowed if we can determine that `f3` is shared.

Nested classes can be annotated `@Thread` and `@Shared` just like regular classes. They can access static variables of the enclosing classes.

Static Fields, Blocks and Methods. In static context, the implicit annotation is `@Shared` rather than `@Context`. Static fields and variables can still be `@Thread`, but never `@Context`. The downside of this design is that static methods used to implement pure functions can never manipulate `@Context` data (see Sec. 5).

Generics and Collections. As stated above, Loci does not yet support Java generics. The reason is that Java does not (yet) support annotations on type parameters. Once Java does, extending Loci to work with generics is straightforward and will mostly follow the style of [28]. On the downside, support for generics will require the introduction of additional annotations to our system to serve as "annotation parameters." The reason is to enable expressing that the annotations on two different types should be bound to the same annotation.

Java's Thread API. As we saw in Fig. 1, `Thread` is a shared class. The same holds for `Runnable`. This is natural, since instances of both will (potentially) be shared between at least two threads. Rather than using thread-local fields in a `Thread` object, a programmer should insert an extra level of indirection pointed to by a `@Thread` local variable in the `run()` method. (This pattern emerged in our evaluation.) As a result of these default annotations, Loci works naturally with Java `ThreadPools`. Java's `InheritableThreadLocal` also works well with the annotations as inherited values are fresh thread-local copies for the new thread. Extending Loci to support `InheritableThreadLocal` is straightforward.

5 Evaluation of Type System Design

To evaluate the type system design and our defaults, we have annotated parts of the JavaGrande benchmark and Lucene Search from the DaCapo Benchmark suite [5]. We have also implemented an inferencer for our system. The results are shown in Tab. 2. In short, they show us that the design of the Loci type system is largely compatible with how Java code is written.

Tab. 2 shows our results from annotating large chunks of Java code. We used code coverage tools to make sure we annotated parts of the code that was actually being executed. We also annotated parts of Xalan, a ≥100 000 LOC XSLT processor, but this work is unfinished at the time of writing. As is visible from the table, the number of annotations is small—80 annotations in total for 44 245 LOC, which is less than 1 annotation per 500 LOC. The annotation of Lucene Search was driven by the desire to only annotate the parts of the code that execute as part of thread-local computation and leave the rest of the code unannotated. We were able to annotate 19 classes as `@Thread` that perform thread-local computation inside instances of `IndexReader`. Due to lack of an annotated Java API, some classes could not be annotated without getting warnings from the Loci tool, notably the index reader itself. Most notably, both classes stored in thread-local fields in the original source (`TermsVectorReader` and `SegmentTermEnum`) could be annotated `@Thread`. The two thread-local fields in Lucene Search in Tab. 2 were uses of `ThreadLocal` that were there from the start and none were added. The two key reasons why a class could not be annotated `@Thread` is because it stores itself in a hash map or extends vector. Both these problems can be solved by annotating the standard library. Raytracer is a much smaller application (only 1496 LOC). Here, all classes could be annotated `@Thread`. There were no uses of `ThreadLocal` to begin with, and none were added.

Though we have not annotated the entire DaCapo suite, we wanted to see what fraction of objects are effectively thread-local. To measure this, we instrumented revision 15.182 of Jikes RVM [21] to report the fraction of live objects

	LOC	Classes	`@Thread`	`@Shared`	Default
Raytracer	1496	16	16	0	0
Lucene Search	42749	285	19	0	266
Total	44245	301	35	0	266

Inferred	Classes
`@Thread`	5996
`@Shared`	1289
Total	7285

	`@Thread` Annotations					`@Shared` Annotations			
	Fields	Params	Returns	Vars		Fields	Params	Returns	Vars
Raytracer	0	0	0	1		1	1	1	1
Lucene Search	2	0	4	5		0	20	0	9
Total	2	0	4	6		1	21	1	10

Average Thread-locality Rate										
				Apache			Lucene			
	ANTLR	BLOAT	Eclipse	FOP	HSQLDB	Jython	Index	Search	PMD	Xalan
Objects	79%	82%	63%	78%	88%	77%	78%	74%	83%	66%
Bytes	71%	77%	64%	76%	85%	73%	71%	51%	81%	69%

Table 2. Results from experiments with annotating Java programs with Loci. The upper right table shows results from applying a conservative analysis to infer annotations to GNU classpath. The bottom shows results of dynamic analysis for DaCapo benchmarks [5]. We measure both the average rate at which objects are thread-local, and the average number of bytes that belong to thread-local objects. In the entire DaCapo suite, 69% of all objects are thread-local.

that have been used from multiple threads. Detecting object accesses was done using a read barrier. These measurements also include objects used by the VM, which itself is heavily multi-threaded, hence even for single-threaded benchmarks like ANTLR, BLOAT, and others, the rate is not 100%. As Tab. 2 shows, all benchmarks have at least half of their heaps occupied by thread-local objects—including heavily multi-threaded ones like Lucene Search. Perhaps unsurprisingly, our results show that small objects tend to be more likely to be thread-local, as evidenced by the rate of object thread-locality being higher than the rate of heap usage by thread-local objects.

Class-Level Annotation Inference. To further test our assumption that most classes can be annotated `@Thread`, we implemented a conservative backwards-flow analysis to detect leakage of **`this`**. Classes that could leak **`this`**, or extended `@Shared` classes, were marked `@Shared`. The remaining classes were marked `@Thread`. Applying the analysis on the GNU Classpath version of the Java standard API, 82% of all classes could be annotated `@Thread`, notably all collection classes. For simplicity, we assumed that native code did *not* leak **`this`**. Assuming native code always leaks, the number is 77% and e.g., all collection classes are shared because of sparse uses of native code in some collection implementations. We have also used our Jikes RVM instrumentation to check that all objects annotated thread-local were indeed accessed by a single thread, and we found that this was the case. Thus, it seems that our implementation is correct.

We now briefly report on the most important realizations from annotating the programs.

Static Methods. Our experiments with Loci shows that our simple defaults-to-@Shared approach for static methods caused problems in many cases where static methods were used as global functions. For example, the Vec class in Raytracer, frequently uses methods like this:

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

Since the a and b in the code about would be @Shared by default from being in a static context, any thread-local vectors are precluded from using this purely functional method. The simple solution for this problem was to simply make these methods instance methods, which was a simple refactoring, but an annoying one. Similar refactorings were done for Lucene Search as well. In the spirit of simplicity, a possible solution to this problem is to allow explicit uses of @Context (they must be explicit to preserve all-shared semantics of unannotated Java programs) on parameters to static methods. The existing type system would prevent leakage as is. A more general but less lightweight solution is a parametric approach using "annotation parameters." This also has use for the problem with equals methods.

Exceptions. The Xalan benchmark uses exceptions to propagate broken XML nodes to a problem reporter. As Exception defaults to shared in our system, this caused a problem for making the XML parsing a thread-local computation as thread-locality would be lost for a broken node wrapped in a shared exception. This practice, and the fact that exceptions cannot propagate into another thread short of being stored on the heap, caused us to rethink this default. We are currently investigating the possibility of annotating Throwable and its subclasses as @Thread and the default annotation on exceptions will be @Context.

Equals Methods. In a @Thread class C, the type of **this** is @Context C. This automatically prevents the leaking of **this** into @Shared variables, but there are also downsides. Consider the typing of Java's equals method. If the parameter to equals has type @Context, then a @Thread class cannot pass **this** to the equals of a shared object, nor vice versa. This turned out to be a rare problem and occurred only twice in Lucene and was solved by ignoring the warnings after having manually inspected the code. Furthermore, we can only compare objects living in the same heap(let), which is unfortunate.

A flexible solution to this problem is supporting annotation-polymorphic methods[1]. This also solves problems with static methods discussed above. An

[1] Since Loci only has one annotation per type, this would not break polymorphism as is the case for full-blown ownership types systems, see [35].

```
 1 @Thread class Foo {                       1 @Thread class Foo {
 2   @Thread Foo foo = null;                 2   @Thread ThreadLocal<Foo> foo = ...
 3   @Thread Foo synchronized m1() {         3   @Thread Foo synchronized m1() {
 4     this.m2();                            4     this.m2();
 5     if (foo != null) m1();                5     if (foo.get() != null) m1();
 6     return foo;                           6     return foo.get();
 7   }                                       7   }
 8                                           8   @Thread Foo Shadow_m1() {
 9                                           9     this.Shadow_m2();
10                                          10     if (foo.get() != null)
11                                          11       Shadow_m1();
12                                          12     return foo.get();
13                                          13   }
14   synchronized void m2() {               14   synchronized void m2() {
15     this.m1();                           15     this.m1();
16   }                                      16   }
17                                          17   void Shadow_m2() {
18                                          18     this.Shadow_m1();
19                                          19   }
20 }                                        20 }
```

Fig. 9. “Ensugared” and “desugared” view of a class with synchronized methods. Desugaring of line 2 is omitted since it is shown in Fig. 8.

alternative solution is to use a different `equals` method for the three possible combinations. As Java does not allow dispatching on annotations, these methods must be differently named, but since which method to use can be statically determined, calls to `equals` can be automatically rewritten under the hood by the tool to use the right version, and the different equals methods automatically inferred from the `@Shared` case. Notably, unless we allow `@Context` to be used explicitly in `@Shared` classes, receiver and argument on equals calls on `@Shared` receivers with `@Context` arguments would have to be switched.

5.1 Removing Unnecessary Synchronization

For flexibility for library classes, `@Thread` classes can be used to create both shared and thread-local objects. This requires extra work to elide locks in Loci. To this end, we introduce a “shadow method,” a duplicate of a method where synchronization on `@Context` objects is removed. Calls on thread-local receivers will call shadow methods, prefixed `Shadow_` if they exist. In shadow methods, **`this`** is thread-local and thus all `@Context` variables are too. Loci creates these methods automatically and transparently. Fig. 9 shows the “ensugared” (standard view) and the “desugared” view of a piece of code. We have implemented this scheme in Loci and tested it on our annotated programs. Without sufficiently annotated Java standard libraries, we will not see any measurable performance benefits due to default-to-`@Shared`. For example, the `IndexReader` class in Lucene Search, which would be key to avoid a fair amount of synchronized methods calls, cannot be annotated `@Thread` due to uses of library use in its methods and methods of its subclasses.

6 Related Work

Domani et al. [17] propose thread-local heaps where each thread is given it own chunk of memory in which to allocate objects. The goal is to remove locking from the GC for thread local objects. They use a dynamic analysis to track thread-locality and do not enforce it. Several researchers have employed compile-time escape analysis to identify local and global objects [6, 7, 10, 32, 11]. These proposals target compiler optimizations (e.g., the removal of unnecessary synchronization) and memory management, and do not support static checking of programmer intentions with respect to thread-locality. Currently, JVMs performs similar analyzes under the hood (see e.g., [8, 20]), but cannot enforce correct usage of `ThreadLocal` or does not give any feedback to the programmer to help verify her programs. Recently, Flanagan et al. [18] extended AtomicJava [19] with support for thread-local data for full-on Java. They target method atomicity and their system is powerful and distinguishes between five different kinds of atomicities. Their system is more powerful than ours and allows any object to act as a guard, whereas we only allow **`this`** to act as a guard for fields. As a result, their system is more complicated and comes at the price of additional complexity and annotation overhead.

Loci is simple ownership type system [24, 15]. Several approaches using ownership types for concurrency control have been proposed [4, 9, 30, 16, 14]. None of these systems use a thread-as-owners approach, nor focuses on thread-local data. Guava [4] presents as an informal collection of rules which would require a significantly more complex type system than the one we present here. StreamFlex [30] use a minimal notion of ownership, with little need for annotations, to simplify memory management in a real-time setting. Cunningham et al. [16] employ Universe Types to “carve up a heap” for safe locking. Their system is similar to ours in that it is based on a simple ownership system [23], but focuses on eliminating data races rather than checking thread-locality. Joëlle [14] proposes a minimal ownership types system in the active objects setting that guarantees that only the thread of an active object will access its representation. The system is built on a different set of principles—sharing is impossible, and all inter-thread communication must be asynchronous or the thread-locality assumption is void. Kilim [31] gives thread-locality through a linear type system for actor-style programming in Java. Kilim replaces copying by transfer of ownership. Sadly, Kilim’s requirement that unique messages be tree-structured (due to linearity) forces regular object structures used as internal representations of communicating actors to be cloned into trees, at least on the sender’s side, before being transferred.

7 Conclusion

We have presented Loci, a simple type system for thread-local data in Java and Java-like languages. We have shown its formal semantics, and stated and proven its crucial properties. Furthermore, we have described our realization of Loci as an Eclipse tool and described how the Loci annotations apply to full-on Java. Experiences with using Loci on known benchmarks showed that the system is compatible with current Java practices, but that further extensions are needed. We will continue to develop Loci while continuing the balance act between simplicity of the annotations, usefulness and legacy integration.

In future work we intend to explore synergies of thread locality information with other optimizations. If a dynamic analysis can make up for its overhead, we envisioning allowing "casts" on the annotations. This will allow a simpler system but also open up for run-time errors. We will also extend Loci to support generics and experiment with the practical usefulness of adding a `@Free` annotation.

Acknowledgments. We thank the ECOOP reviewers for their suggestions which helped improve the presentation of this paper. The impetus for this work came from discussion with Doug Lea. The paper benefited from comments and discussion with Johan Östlund. The authors were partially supported by grants NSF CPA 0811631, NSF CPA 0811691 and NSF 0720652.

References

1. Java theory and practice: Synchronization optimizations in mustang. http://www-128.ibm.com/developerworks/java/library/j-jtp10185/.
2. J. Aldrich, C. Chambers, E. G. Sirer, and S. J. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *SAS*, pages 19–38, 1999.
3. D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *PLDI*, pages 258–268, 1998.
4. D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *OOPSLA*, pages 382–400, 2000.
5. S. M. Blackburn and others. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.
6. B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA*, pages 20–34, 1999.
7. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA*, pages 35–46, 1999.
8. S. Borman. Sensible sanitation – understanding the IBM Java garbage. IBM DeveloperWorks, August 2002.
9. C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.
10. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

11. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.
12. D. Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, 2001.
13. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, 2003.
14. D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen. Minimal Ownership for Active Objects. Technical Report SEN-R0803, CWI, 2008.
15. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
16. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, Sept. 2007.
17. T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *ISMM*, pages 76–87, 2002.
18. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM TOPLAS*, 30(4):1–53, 2008.
19. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349, 2003.
20. B. Goetz. Java theory and practice: Urban performance legends, revisited. IBM DeveloperWorks, September 2005.
21. Jikes RVM homepage. http://jikesrvm.org/.
22. P. G. Joisha. Compiler optimizations for nondeferred reference: counting garbage collection. In *ISMM*, pages 150–161, 2006.
23. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, Springer, 2002.
24. J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection, In *ECOOP*, pages 158–185, 1998.
25. OpenJDK. http://openjdk.java.net/.
26. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness and immutability. In *TOOLS*, 2007.
27. F. Pizlo, A. L. Hosking, and J. Vitek. Hierarchical real-time garbage collection. In *LCTES*, pages 123–133, 2007.
28. A. Potanin. *Generic Ownership—A Practical Approach to Ownership and Confinement in OO Programming Languages.* PhD thesis, Victoria University of Wellington, 2007.
29. K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *OOPSLA*, pages 263–272, 2006.
30. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput Stream Programming in Java. In *OOPSLA*, pages 211–228, 2007.
31. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for Java. In *ECOOP*, pages 104–128, 2008.
32. B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM*, pages 18–24, 2000.
33. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
34. T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Kista, Stockholm, 2006.
35. T. Wrigstad and D. Clarke. Existential owners for ownership types. *Journal of Object Technology*, 4(6):141–159, 2007.