Type Inference for Parameterized Race-Free Java[⋆]

Rahul Agarwal and Scott D. Stoller

Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400

Abstract. We study the type system introduced by Boyapati and Rinard in their paper "A Parameterized Type System for Race-Free Java Programs" and try to infer the type annotations ("lock types") needed by their type checker to show that a program is free of race conditions. Boyapati and Rinard automatically generate some of these annotations using default types and static inference of lock types for local variables, but in practice, the programmer still needs to annotate on the order of 1 in every 25 lines of code. We use run-time techniques, based on the lock-set algorithm, in conjunction with some static analysis to automatically infer most or all of the annotations.

1 Introduction

Type systems are well established as an effective technique for ensuring at compile-time that programs are free from a wide variety of errors. New type systems are being developed by researchers at an alarming rate. Many of them are very elaborate and expressive.

Types provide valuable compile-time guarantees, but at a cost: the programmer must annotate the program with types. Annotating new code can be a significant burden on programmers. Annotating legacy code is a much greater burden, because of the vast quantity of legacy code, and because a programmer might need to spend a long time studying the legacy code before he or she understands the code well enough to annotate it.

Type inference reduces this burden by automatically determining types for some or all parts of the program. A type inference mechanism is *complete* if it can infer types for all typable programs.

Traditional type inference is based on static analysis. A common approach is constraint-based type inference, which works by constructing a system of constraints (of appropriate forms) that express relationships between the types of different parts of the program and then solving the resulting constraints.

Unfortunately, complete type inference is impossible or infeasible for many expressive type systems. This motivates the development of incomplete type inference algorithms. These algorithms fall on a spectrum that embodies a trade-off

[⋆] This work was supported in part by NSF under Grant CCR-9876058 and ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Authors' Email: {ragarwal,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{ragarwal,stoller}

between computational cost and power. Roughly speaking, we measure an algorithm's power by how many annotations the user must supply in order for the algorithm to succesfully infer the remaining types. For some important type systems, even incomplete algorithms designed to infer most types for most programs encountered in practice may have prohibitive exponential time complexity.

We have developed a new run-time approach to type inference that, for some type systems, appears to be more effective in practice than traditional static type inference. We monitor some executions of the program and infer (one might say "guess") candidate types based on the observed behavior.

A premise underlying this work is that data from a small number of simple executions is sufficient to infer most or all of the types. In particular, it is *not* necessary for the monitored executions collectively to achieve—or even come close to—full statement coverage in order to support successful inference of types for the entire program. Another premise is that the type inference is relatively insensitive to the choice of test inputs. Experience with Daikon [Ern00] supports this idea.

This approach has some obvious theoretical limitations. It is not complete, because the process of generalizing from relationships between specific objects in a particular execution to static relationships between expressions or statements in the program is based in part on (incomplete) heuristics. Run-time type inference is unsound (*i.e.*, the inferred types might not satisfy the type checking rules), because the observed behavior is not necessarily characteristic of all possible behaviors of the program, and correct types express properties that should hold for all possible behaviors of the program. We transform this unsoundness into incompleteness by running the type-checker after type inference. If the type-checker signals an error, we report that type inference failed. This is not a problem in practice, provided it is rare.

Despite these theoretical limitations, from a pragmatic point of view, the most important question for a given type system is whether run-time type inference, traditional static type inference, or a combination of them will provide the greatest overall reduction of users' annotation burden. As a first step towards the empirical evaluation of run-time type inference, we developed and implemented a run-time type inference algorithm for a recently proposed type system for concurrent programs.

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. This inspired the development of type and effect systems (for brevity, we will call them "type systems" hereafter) that statically ensure the absence of some common kinds of concurrent programming errors. Flanagan and Freund [FF00] developed a type system that ensures that a Java program is race-free, *i.e.*, contains no race conditions; a *race condition* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write. The resulting programming language (*i.e.*, Java with their extensions to the type system) is called *Race Free Java*. Boyapati and Rinard [BR01] modified and extended Flanagan and Freund's type system to make it more expressive. The resulting programming language is called *Param-*

eterized Race Free Java (PRFJ). Hereafter, we assume that programs contain all type information required by the standard Java type-checker, and we use the word "types" to refer only to the *additional* type information required by these extended type systems.

These type systems encode programming patterns that experienced programmers often use to avoid these errors. Specifically, these type systems track the use of locks to "protect" (*i.e.*, prevent concurrent access to) shared data structures. Although these type systems are occasionally undesirably restrictive, experience indicates that they are sufficiently expressive for many programs.

The cost of expressiveness is that type inference for these type systems is difficult. In Flanagan and Freund's experiments with three medium or large programs, well-chosen defaults, in combination with some potentially unsafe (but usually safe in practice) "escapes" from the type system, reduce the number of annotations needed to about 12 annotations/KLOC (KLOC denotes "thousand lines of code") on average [FF00]. For a fair comparison with type inference for PRFJ (discussed below), note that PRFJ programs typically require more annotations than this, primarily because the PRFJ type system is more expressive and does not rely on potentially unsafe escapes.

Flanagan and Freund subsequently developed a simple type inference algorithm for their type system. Roughly speaking, it starts with a set of candidate types for each expression, runs the type checker, deletes some of the candidate types based on the errors (if any) reported by the type checker, and repeats this process until the type checker reports no errors [FF01]. However, this algorithm infers types only for a restricted version of the type system—specifically, a version without external locks. Elimination of external locks significantly reduces the expressiveness of the type system.

Boyapati and Rinard [BR01] use carefully-chosen defaults and local type inference to reduce the annotation burden on users of PRFJ. The user provides type annotations on selected declarations of classes, fields, and methods, and selected object allocation sites (*i.e.*, calls to `new`). Default types are used where annotations are omitted. A simple intra-procedural constraint-based type inference algorithm is used to infer types for local variables of each method. In their experiments with several small programs, users needed to supply about 25 annotations/KLOC [BR01].

We believe that type systems like PRFJ are a promising practical approach to verification of race-freedom for programs that use locks for synchronization, except that the annotation burden is currently too high. We attempted to develop static inter-procedural type inference algorithms for PRFJ, using abstract interpretation [CC77] and constraint-based analysis (*e.g.*, instantiation constraints [AKC02]), but the analysis algorithms were computationally expensive, and we did not believe they would scale to large programs.

This motivated us to develop and implement a run-time type inference algorithm for PRFJ. The program is instrumented by an automatic source-to-source transformation. The instrumented program writes relevant information (mainly information about which locks are held when various objects are accessed) to a log file. Analysis of the log, together with a simple static program analysis that identifies unique pointers [AKC02], produces type annotations at selected program points. Boyapati and Rinard's simple intra-procedural type inference algorithm is then used to propagate the resulting types to other program points. This has the crucial effect of propagating type information into branches of the program that were not exercised in the monitored executions. Our experience, reported in detail in Section 6, is that run-time type inference provides a significant further reduction in the annotation burden.

If multiple types for (say) a field declaration are consistent with the logged information, we use heuristics to prioritize them. If the candidate type with the highest priority is rejected by the type checker (because it is inconsistent with the types at other program points), we try the next one. In our experiments so far, the highest-priority choice has always worked, so this iterative approach has not been needed (or implemented).

Future Work. In the short term, we plan to implement a type checker for PRFJ; our current lack of a type checker limited our experiments to programs small enough for us to type-check manually. The run-time overhead of our instrumentation is already moderate (see Section 6) but could perhaps be reduced by incorporating ideas from [vPG01]. In the longer term, we are investigating type inference for race-free variants of C, such as [Gro03]. We expect run-time type inference to be even more beneficial in this context. Boyapati and Rinard's defaults in PRFJ are very effective, in part because Java provides built-in locks, so `this` is a very good guess at the identity of the protecting lock in many cases. Devising equally effective defaults for variants of C seems difficult.

2 Related Work

Run-time type inference is very similar in spirit to Daikon [Ern00]. During execution of a program, Daikon evaluates a large syntactic class of predicates at specified program points and determines, for each of those program points, the subset of those predicates that always held at that program point during the monitored executions. Among those predicates, those that satisfy some additional criteria are reported as candidate invariants. Daikon cannot infer PRFJ types, because the invariants expressed by PRFJ types are not expressible in Daikon's language for predicates. Daikon infers predicates that can be evaluated at a single program point. In contrast, a single PRFJ type annotation can express an invariant that applies to many program points. For example, if the declaration of a field f in a class C is annotated with the PRFJ type `self`, it means (roughly): for all instances o of class C, for all objects o' ever stored in field f of o, o' is protected by its own lock, *i.e.*, the built-in lock associated (by the Java language semantics) with o' is held whenever any field of o' is accessed. Such accesses may occur throughout the program.

Naik and Palsberg developed an expressive type system for ensuring that an interrupt-driven program will handle every interrupt within a specified deadline

[NP03]. Their type system is equivalent to model checking, in the sense that a program is typable exactly when their model checker, applied to a specified abstraction of the program, verifies that all deadlines are met. Due to this close connection, types for a program can be inferred from the output of the model checker. Our goal is quite different than theirs: we aim to show that inexpensive run-time techniques (in contrast to relatively expensive model checking) can provide an effective basis for type inference.

Static analyses such as meta-compilation [HCXE02] and type qualifiers [FTA02] have been used to check or verify simple lock-related properties of concurrent programs, *e.g.*, that a lock is not acquired twice by the same thread without an intervening release. Such analyses cannot easily be used to check more difficult properties such as race-freedom.

3 Overview of Parameterized Race Free Java (PRFJ)

The PRFJ type system is based on the concept of object ownership. Each object is associated with an owner which is specified as part of the type of the variables that refer to that object. Each object is owned by another object, or by special values `thisThread`, `self`, `unique` or `readonly`. Since an object can be owned by another object which in turn could be owned by another object, the ownership relation can be regarded as a forest of rooted trees, where the roots may have self loops. Ownership information expresses a synchronization discipline: to safely access an object o, a thread must hold the lock associated with the root r of the ownership tree containing o; r is called o's *root owner*.

An object with root owner `thisThread` is unshared. Such objects can be accessed without synchronization. This is reflected in the type system by declaring that every thread implicitly holds the lock associated with `thisThread`. An object with owner `self` is simply owned by itself. If an object o has owner `unique`, there is a single (unique) reference to o. Only the thread currently holding that reference can access o, so there is no possibility of race conditions involving o, and no lock needs to be held when accessing o. An object with owner `readonly` cannot be updated and can be accessed without any locks.

Every class in PRFJ is parameterized with one or more parameters. Parameterization allows the programmer to specify appropriate ownership information separately for each use of the class. The first parameter always specifies the owner of the `this` object. The remaining parameters, if any, may specify the owners of fields or parameters or return values of methods. The first parameter of a class can be a formal owner parameter or one of the special values discussed above; the remaining parameters must be formal owner parameters. When the class is used in the program, its formal owner parameters are instantiated with final expressions or the above special values. *Final expressions* are expressions whose value does not change; using them to represent owners ensures that an object's owner does not change from one object to another. Syntactically, final expressions are built from final variables, including the implicit `this` variable, final fields, and static final fields. Ownership changes that do not lead to race

```
public class MyThread<thisThread> extends Thread<thisThread> {

  public ArrayList<self,readonly> ls;

  public MyThread(ArrayList<self,readonly> ls) {
    this.ls = ls;
  }

  public void run() {
   synchronized(this.ls) { ls.add(new Integer<readonly>(10)); }
  }

 public static void main(String args[]) {
  ArrayList<self,readonly> ls = new ArrayList<self,readonly>();
  MyThread<unique> m1 = new MyThread<unique>(ls);
  MyThread<unique> m2 = new MyThread<unique>(ls);

  m1--.start();
  m2--.start();
 }
}
```

Fig. 1. A Sample PRFJ Program.

conditions are allowed; for example, an object's owner may change from `unique` to any other owner.

Every method is annotated with a clause of the form "`requires` $e_1, \ldots, e_n$, where the e_i are final expressions. Locks on the root owners of the objects listed in the `requires` clause must be held at each call site.

The type checking rules ensure that in a well-typed program, an object that is not readonly can be accessed only by a thread that either holds the lock on the root owner of the object or has a unique reference to the object. This implies that the program is race-free.

To illustrate the PRFJ system, consider the program in Figure 1. The definition of class `ArrayList` is not shown, but it has two owner parameters: the first specifies the owner of the `ArrayList` itself, and the second specifies the owner of the objects stored in the `ArrayList`. The `MyThread` constructor returns a unique reference to the newly allocated object. Thus, the main thread has unique references to the two instances of `MyThread` until they are started. After an instance of `Mythread` is started, it is accessed by only one thread (namely, itself) and hence is unshared. Thus, the owner of each `MyThread` object changes from `unique` to `thisThread`. The occurrences of `--` in the `main` method indicate that the main thread relinquishes its unique references to `m1` and `m2` when it starts them. These occurrences of `--` are required by the type checking rules, and we consider them to be, in effect, type annotations.

The two instances of `MyThread` share a single `ArrayList` object *a*. The lock associated with *a* is held at every access to *a*, so *a* has owner `self`, and the first parameter of `ArrayList` is instantiated with `self`. Instances of the `Integer` class are immutable, so they have owner `readonly`. All objects stored in *a* have owner `readonly`, so the second owner parameter of `ArrayList` is instantiated with `readonly`.

PRFJ defaults are unable to determine the `unique`, `self`, and `readonly` owners used in this program. Our type-inference algorithm described in Section 4 infers all of the types correctly for this program. [AS03] shows in detail how our algorithm works for this program.

4 Type Inference for PRFJ

Our algorithm has three main steps.

First, the static analysis in [AKC02] is used to infer `unique` and `!e` (" not escaping") annotations for fields, method parameters, return values, and local variables (PRFJ's *!e* annotation corresponds to the `lent` annotation in [AKC02]). We use static analysis for this because it is usually adequate and because run-time determination of which objects have unique references would be expensive.

Second, run-time information is used to infer owners for fields, method parameters and return values. Owners in class declarations are inferred next. For a class whose first owner is inferred to be a constant (*i.e.*, anything other than a formal owner parameter), all occurrences of that class in the program are instantiated with that constant as the first owner.

Third, the intra-procedural type inference algorithm in [BR01, Section 7.1] is applied, to infer the types of local variables whose types have not already been determined.

Few classes need multiple owner parameters, and most of the classes that do are library classes, which can be annotated once and re-used, so we do not attempt to infer which classes C need multiple owner parameters or how those parameters should be used in the declarations of fields and methods of C. We assume this information is given. We do try to infer how to instantiate those owner parameters in all uses of C.

4.1 Inferring Unique Owners

The static uniqueness analysis in [AKC02] is a fairly straightforward flow-sensitive context-insensitive inter-procedural data-flow analysis whose running time is linear in the size of the program. For details, see [AKC02].

4.2 Inferring Owners for Fields, Method Parameters and Return Values

Let x denote a field, method parameter, or method return type with reference type (*i.e.*, not a base type). To infer the owner of x (*i.e.*, the first-owner in the type of x), we monitor accesses to a set $S(x)$ of objects associated with x. If x is a field of some class C, $S(x)$ contains objects stored in the f field of instances of C. For a method parameter x, $S(x)$ contains arguments passed through that parameter. For a method return type x, $S(x)$ contains objects returned from the method. Let $\mathrm{FE}(x)$ denote the set of final expressions that are syntactically legal (*i.e.*, in scope) at the program point where x is declared.

After an object o is added to $S(x)$, every access (read or write) to o is intercepted and some information is recorded. Specifically, at the end of run-time monitoring, the following information is available for each object o in $S(x)$: $\mathrm{lkSet}(x, o)$, the set of locks that were held at every access to o after o was inserted in $S(x)$ [SBN+97]; $\mathrm{rdOnly}(x, o)$, a boolean that is true iff no field of o was written (updated) after o was inserted in $S(x)$; $\mathrm{shar}(x, o)$, a boolean that is true iff o is "shared", *i.e.*, multiple threads accessed non-final fields of o after o was inserted in $S(x)$; $\mathrm{val}(x, o, e)$, the value of final expression e at an appropriate point for x and o, for each $e \in \mathrm{FE}(x)$. If x is a field, the appropriate point is immediately after the constructor invocation that initialized o. If x is a parameter of a method m, the appropriate point is immediately before calls to m at which o is passed through parameter x. If x is a return type of a method m, the appropriate point is immediately after calls to m at which o is passed as the return value of m.

The owner of x is determined by the first applicable rule below.

1. If the Java type of x is an immutable class (*e.g.*, `String` or `Integer`), then $\mathrm{owner}(x)$=`readonly`.
2. If $(\forall o \in S(x) : \neg\mathrm{shar}(x, o))$, then $\mathrm{owner}(x)$=`thisThread`.
3. If $(\forall o \in S(x) : \mathrm{rdOnly}(x, o))$, then $\mathrm{owner}(x)$=`readonly`.
4. If $(\forall o \in S(x) : o \in \mathrm{lkSet}(x, o))$, then $\mathrm{owner}(x)$=`self`.
5. Let $E(x)$ be the set of final expressions e in $\mathrm{FE}(x)$ such that, for each object o in $S(x)$, $\mathrm{val}(x, o, e)$ is a lock that protects o; that is, $E(x) = \{e \in \mathrm{FE}(x) \mid \forall o \in S(x) : \mathrm{val}(x, o, e) \in \mathrm{lkSet}(x, o)\}$. If $E(x)$ is non-empty, take $\mathrm{owner}(x)$ to be an arbitrary element of $E(x)$.
6. Take $\mathrm{owner}(x)$ to be a formal owner parameter of the class containing x, normally `thisOwner`.[1]

To reduce the run-time overhead, we restrict $S(x)$ to contain only selected objects associated with x. This typically does not affect the inferred types. We currently use the following heuristics to restrict $S(x)$. For a field x with type C, $S(x)$ contains at most one object created at each allocation site for C. For a method parameter or return type x, $S(x)$ contains at most one object per call site of that method. Also, we restrict $\mathrm{FE}(x)$ to contain only the values of final expressions of the form `this` or `this.`f, where f is a final field.

4.3 Inferring Values of Non-First Owner Parameters

If the Java type of x is a class C with multiple owner parameters, for each formal owner parameter P of C other than the first, we need to infer $\mathrm{owner}_P(x)$,

[1] This rule is not needed for the examples in Section 6.

the value with which P should be instantiated for x. Let $S_P(x)$ denote a set of objects o' associated with P for x and such that P denotes the first owner of o'. In particular, for each o in $S(x)$: (1) for each field f of C declared with P as the first owner of f (*i.e.*, `class` C<..., P, ...> `{ ...` D<P> f`; ... }`), objects stored in $o.f$ are added to $S_P(x)$; (2) for each parameter p of a method m of C such that the first owner of p is P (*i.e.*, `class` C<..., P, ...> `{ ...` m`(...,`D<P> p`,...) ... }`), add to $S_P(x)$ arguments passed through parameter p when $o.m$ is invoked; (3) for each method m of C whose return type has first owner P, add to $S_P(x)$ objects returned from invocations of $o.m$. We instrument the program to monitor accesses to objects in $S_P(x)$ and infer an owner based on that, just as in Section 4.2. As an optimization, we may restrict $S_P(x)$ to contain a subset of the objects described above.

4.4 Inferring Owners in Class Declarations

Let owner(C) denote the first owner in the declaration of class C (*i.e.*, it denotes o in `class` C<o`,...`>). Let $S(C)$ contain instances of C. We monitor accesses to elements of $S(C)$ as in Section 4.2 and then use the following rules to determine owner(C).

1. If C is a subclass of a class C' with owner(C')=`self`, then owner(C)=`self`.
2. If $S(C) = \emptyset$ (*i.e.*, there are no instances of C), then owner(C)=`thisOwner`.[2]
3. If $(\forall o \in S(C) : \neg\text{shar}(x, o))$, then owner($C$)=`thisThread`.
4. If $(\forall o \in S : o \in \text{lkSet}(x, o))$, then owner($C$)=`self`.
5. Take owner(C)=`thisOwner`.

For efficiency, we restrict $S(C)$ to contain only a few instances of C. Currently, we arbitrarily pick two fields or method parameters or return values of type C and take $S(C)$ to contain the objects stored in or passed through them.

4.5 Inferring `requires` Clauses

We infer `requires` clauses basically as in [BR01], except we use run-time monitoring instead of user input to determine which classes C have owner `thisThread` (*i.e.*, each instance of C is accessed by a single thread).

Each method declared in each class with owner `thisThread` is given an empty `requires` clause. For each method in each other class, the `requires` clause contains all method parameters p (including the implicit `this` parameter) such that m contains a field access $p.f$ (for some field f) outside the scope of a `synchronized(`p`)` statement; as an exception, the `run()` method of classes that implement `Runnable` is given an empty `requires` clause, because a new thread holds no locks.

[2] For example, in many programs, the class containing the `main` method is never instantiated.

4.6 Static Intra-Procedural Type Inference

The last step is to infer the types of local variables whose types have not already been determined, using the intra-procedural type inference algorithm in [BR01, Section 7.1]. Each incomplete type (*i.e.*, each type for which the values of some owner parameters are undetermined) is filled out with an appropriate number of fresh distinct owner parameters. Equality constraints between owners are constructed in a straightforward way from each assignment statement and method invocation. The constraints are solved in almost linear time using the standard union-find algorithm. For each of the resulting equivalence classes E, if E contains one known owner o (*i.e.*, an owner other than the fresh parameters), then replace the fresh owner parameters in E with o. If E contains multiple known owners, then report failure. If E contains only fresh owner parameters, then replace them with `thisThread`. This heuristic is adequate for the examples we have seen, but if necessary, we could instrument the program to obtain runtime information about objects stored in those local variables, and then infer their owners as in Section 4.2.

5 Implementation

This section describes the source-to-source transformation that instruments a program so that it will record the information needed by the type inference algrithm in Section 4. The transformation is parameterized by the set of classes for which types should be inferred.

All instances of `Thread` are replaced with `ThreadwithLockSet`, a new class that extends `Thread` and declares a field `locksHeld`. Synchronized statements and synchronized methods are instrumented to update `locksHeld` appropriately; a `try/finally` statement is used in the instrumentation to ensure that exceptions are handled correctly. For each field, method parameter and return type x being monitored, a distinct `IdentityHashMap` is added to the source code. The hashmap for x is a map from objects o in $S(x)$ to the information recorded for o, as described in Section 4, except the lockset. We store all locksets in a single `IdentityHashMap`. Thus, even if an object o appears in $S(x)$ for multiple x, we maintain a single lockset for o. Object allocation sites, method invocation sites, and field accesses are instrumented to update the hashmaps appropriately. Which sites and expressions are instrumented depends on the set of classes specified by the user.

6 Experience

To evaluate the inference engine, we ran our system on the five multi-threaded server programs used in [BR01]. C. Boyapati kindly sent us the annotated PRFJ code for these servers. The programs are small, ranging from about 100 to 600 lines of code. We compare the number of annotations in that code—this is also the number of annotations needed for the mechanisms in [BR01] to successfully infer the remaining types—with the number of annotations needed with our type-inference algorithm. Recall that our type-inference algorithm is also incomplete and hence might be unable to infer some types. In these experiments, we infer types only for the application (*i.e.*, server) code; we assume PRFJ types are given for Java API classes used by the servers.

In summary, our type-inference mechanism successfully inferred complete and correct typings for all five server programs, with no user-supplied annotations. Also, slowdown due to the instrumentations was typically about 20% or less.

Four of the server programs did not come with clients, so we wrote very simple clients for them. One server program (PhoneServer) came with a simple client. We modified the servers slightly so they terminate after processing one or two requests (in our current implementation, termination triggers writing of collected information to the log). A single execution of each server with its simple client provided enough information for successful run-time type inference. This supports our conjecture (in Section 1) that data from a small number of simple executions is sufficient.

The original PRFJ code for GameServer requires 7 annotations. Our algorithm infers types for the program with no annotations. We wrote a simple client with two threads. This program is sensitive to the scheduling of threads. Different interleavings of the threads caused different branches to be taken, leaving other branches in the code unexercised in that execution. We applied our run-time type inference algorithm to each of the possible executions, and it sucessfully inferred the types in every case.

The original ChatServer code contains 13 annotations. We wrote a simple client with two threads. One class (`read_from_connection`) in the server program has only final fields, and the class declaration can correctly be typed with owner `self` or `thisThread`. The original code contains a manual annotation of `self`, while our algorithm infers `thisThread`. The ChatServer, like the other servers, uses Boyapati's modified versions of Java API classes (*e.g.*, `Vector`), from which synchronization has been removed. The benefit is that synchronization can be omitted in contexts where it is not needed; the downside is that, when synchronization is necessary, it must be included explicitly in the application code. We also considered a variant of this server that uses unmodified Java library classes. Our algorithm infers a complete and correct typing for both variants, with no assistance from the user.

The original QuoteServer code contains 15 annotations, PhoneServer code contains 12 annotations across 4 classes and HTTPServer contains 23 annotations across 7 classes. Our algorithm infers types for each of these programs with no annotations. Most of the annotations in QuoteServer were unique annotations, and static analysis of uniqueness is able to infer the annotations. For the HTTPServer program the inferred types are not exactly the same as those in the original code, as was the case with `ChatServer`.

Encouraged by these initial results, we plan to apply our system to much larger examples as soon as we have implemented a type-checker for PRFJ.

Acknowledgement. We thank Chandra Boyapati for many helpful comments about PRFJ.

References

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 2002.

[AS03] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. Technical Report DAR 03-10, Computer Science Department, SUNY at Stony Brook, October 2003.

[BR01] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, 2001.

[CC77] Patrick Cousot and Radhia Cousot. A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington, Department of Computer Science and Engineering, 2000.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[FF01] Cormac Flanagan and Stephen Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96. ACM Press, June 2001.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002.

[Gro03] Dan Grossman. Type-safe multithreading in Cyclone. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25. ACM Press, 2003.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82. ACM Press, 2002.

[NP03] Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. Master's thesis, Purdue University, 2003.

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, October 2001.