

EFFECTIVE STATIC RACE DETECTION FOR JAVA

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Mayur Hiru Naik

March 2008

© Copyright by Mayur Hiru Naik 2008
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Alex Aiken) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dawson Engler)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Monica S. Lam)

Approved for the University Committee on Graduate Studies.

Abstract

Concurrent programs are notoriously difficult to write and debug, a problem that is becoming acute with the recent shift in hardware from uniprocessors to multicore processors. A fundamental concurrency bug is a *race*: a condition in a shared-memory multithreaded program in which a pair of threads may access the same memory location without any ordering enforced between the accesses, and at least one of the accesses is a write. Despite thirty years of research on race detection, today's concurrent programs are still riddled with harmful races.

We present an effective approach to static race detection for Java. We dissect the specification of a race to identify four natural conditions, each of which is sufficient for proving a given pair of accesses race-free, and all of which are necessary in practice as different pairs of accesses may be race-free for different reasons. We present four static analyses each of which conservatively approximates a separate condition while together enabling the overall algorithm to report a useful set of potential races. We have implemented our approach and report upon our experience applying it to a suite of eight multithreaded Java programs which includes a mix of libraries, complete programs, previously studied programs, and newer, real-world, open-source programs.

For complete programs, the approach is sound in that it finds all races. On our benchmark suite, the approach is precise in that it has a false positive rate of 25% (only one in every four reported races is not in fact a race) and it is reasonably scalable in that it is fully automatic and checks programs comprising hundreds of thousands of Java bytecodes in a few minutes. Finally, the approach is effective, finding tens to hundreds of previously unknown concurrency bugs in mature and widely used Java programs in our benchmark suite, many of which were fixed upon reporting.

Acknowledgments

First and foremost, I am grateful to my adviser Alex Aiken for his guidance and help on all aspects of doing research, ranging from how to select the right problems to how to effectively communicate one's solutions.

I thank Dawson Engler, Monica Lam, and Henny Sipma for taking the time to serve on my thesis committee and providing useful feedback.

I am grateful to John Whaley for enthusiastically answering my innumerable questions about his work on BDD-based program analysis upon which my thesis work is based.

I would like to thank my office mates Brian and Yichen, and other colleagues at Stanford including Cristian, Junfeng, Peter, Sorav, and Suhabe for their camaraderie. I also thank my friends Anish, Dilys, Krishnaram, Kristina, Parag, Penka, Rajat, Rajiv, Siddharth, Utkarsh, and Vijay for making my time at Stanford very memorable.

I thank Jens Palsberg, my M.S. thesis adviser at Purdue, for introducing me to the field of programming languages and software engineering. I also thank Jim Larus, Tom Ball, Sriram Rajamani, and Jakob Rehof for teaching me much about program analysis during two summer internships at Microsoft Research.

Finally, I thank my family for their support and my wife Sneha for her unconditional love. I dedicate this thesis to them.

Contents

Abstract	v
Acknowledgments	vi
1 Introduction	1
1.1 Motivation	1
1.2 Challenges	2
1.3 Our Approach	3
1.4 Summary of Contributions	6
1.5 Organization	7
2 Basic Race Detection Algorithm	9
2.1 Introduction	9
2.2 Example	12
2.3 Preprocessing	14
2.4 k -Object-Sensitive Analysis	15
2.4.1 Computation of <code>originalRaces</code>	22
2.4.2 Computation of <code>aliasingRaces</code>	27
2.5 Thread Escape Analysis	29
2.5.1 Intraprocedural Analysis	31
2.5.2 Interprocedural Analysis	33
2.5.3 Computation of <code>escapingRaces</code>	34
2.6 May-Happen-In-Parallel Analysis	37
2.6.1 Computation of <code>parallelRaces</code>	38

2.7	Lockset Analysis	41
2.7.1	Computation of <code>unlockedRaces</code>	43
2.8	Putting it all together	45
2.9	Usability Issues	48
2.9.1	Counterexamples	49
2.9.2	Open Programs	51
2.10	Unsoundness	54
2.11	Experiments	54
2.12	Related Work	59
2.12.1	Dynamic Race Detection	59
2.12.2	Static Race Detection	60
2.12.3	Atomicity Checking	62
3	Conditional Must Not Aliasing	64
3.1	Introduction	64
3.2	Example	67
3.3	Language	71
3.3.1	Syntax	72
3.3.2	Semantics	72
3.4	Type and Effect System	76
3.5	Disjoint Reachability Analysis	85
3.6	Practical Considerations	89
3.7	Conditional Must Not Alias Analysis	90
3.7.1	Computation of <code>globalUnlockedRaces</code>	93
3.7.2	Computation of <code>localUnlockedRaces</code>	94
3.7.3	Computation of <code>threadUnlockedRaces</code>	97
3.8	Experiments	97
3.9	Related Work	100
4	A Demand-Driven Approach	103
4.1	Introduction	103
4.2	Example	105

4.3	Language	113
4.4	k -Object-Sensitive Analysis	116
4.5	Demand-Driven Race Detection Algorithm	123
4.6	Experiments	130
4.7	Related Work	135
5	Conclusion	138
5.1	Future Work	139
A	Proof of Type Preservation	141

List of Tables

2.1	Races in example program.	25
2.2	Benchmarks.	54
2.3	Experimental results: Running time.	55
2.4	Experimental results: Numbers of race pairs.	56
2.5	Experimental results: Classification of reported races.	56
3.1	Benchmarks.	98
3.2	Experimental results: Comparison of running time.	99
3.3	Experimental results: Comparison of numbers of unlocked race pairs.	100
3.4	Experimental results: Comparison of numbers of ultimate race pairs.	100
4.1	Benchmarks.	130
4.2	Experimental results: Comparison of running time.	131
4.3	Experimental results: Number of sites in Σ after last iteration.	132
4.4	Experimental results: Comparison of numbers of abstract contexts.	133
4.5	Experimental results: Comparison of numbers of ultimate race pairs.	134

List of Figures

2.1	Example multithreaded Java program.	13
2.2	k -object-sensitive analysis of example program using $k = 2$	19
2.3	May-happen-in-parallel analysis of example program.	40
2.4	Example multithreaded Java program accessing a static field.	41
2.5	Lockset analysis of example program.	42
2.6	Example harness.	53
3.1	Example multithreaded Java program.	68
3.2	Data structure created by example program.	68
3.3	Abstract syntax of WHILE language.	72
3.4	Semantic domains of WHILE language.	73
3.5	Instrumented operational semantics of WHILE language.	74
3.6	Syntax of types and effects.	77
3.7	Abstraction relations.	78
3.8	Type rules.	80
3.9	Disjoint reachability property and disjoint reachability analysis. . . .	84
4.1	Example multithreaded Java program.	106
4.2	Data structure built by example program.	108
4.3	Heap abstractions computed for example program.	109
4.4	Abstract syntax.	114
4.5	Analysis domains.	116
4.6	k -object-sensitive analysis.	118
4.7	Computation of $\mathcal{A}$	122

4.8	Demand-driven static race detection algorithm.	124
-----	--	-----

Chapter 1

Introduction

1.1 Motivation

Concurrent software is ubiquitous. Concurrency is the key to effective responsiveness, resource utilization, and throughput in software we interact with routinely, such as operating systems, web servers, databases, GUI applications, and games.

Concurrency is expected to become even more pervasive with the recent shift in hardware from uniprocessors to multicore processors [66, 80]. Uniprocessor speeds peaked around the beginning of 2003 due to physical issues such as heat dissipation and power consumption. Since then, all major processor vendors have begun manufacturing multicore processors, which consist of two or more independent cores packed onto a single chip. Moreover, the number of cores in multicore processors is expected to grow exponentially, quite like uniprocessor speeds increased exponentially until recently. This shift in hardware has major implications for software. In the past, as uniprocessor speeds increased, even sequential programs ran faster without any modification. In the future, however, as the number of cores in multicore processors increases, primarily concurrent programs will be able to gain their performance benefits. As a result, future software is expected to be increasingly written in a concurrent fashion.

The benefits of concurrency, however, are counterbalanced by the notorious difficulty of writing and debugging concurrent programs. A fundamental and particularly

insidious concurrency bug is a *race*: a condition in a shared-memory multithreaded program in which a pair of threads may access the same memory location without any ordering enforced between the accesses, and at least one of the accesses is a write. While some races are benign, many are harmful violations of program invariants. In the extreme, races can be disastrous, such as those deemed responsible for the failure of the Therac-25 radiation therapy machine that led to the death of five patients and injured several more, and the breakdown of the Energy Management System that led to the North American Blackout of 2003. Despite three decades of work on race detection, however, today's concurrent programs are still riddled with harmful races.

1.2 Challenges

Races are typically triggered under very specific thread schedules, and the inherent non-determinism of thread schedules renders races not only more likely to elude detection by prevalent industrial testing techniques, but also more difficult to reproduce and fix once they have been detected. As a result, there has been a considerable amount of work on tools for race detection.

Current race detection tools are predominantly based on dynamic (run-time) analysis. State-of-the-art dynamic race detectors are precise and scalable. Like any dynamic analysis, however, they are inherently unsound and cannot be applied to incomplete programs such as libraries.

Race detection tools based on static (compile-time) analysis typically sacrifice some combination of soundness, precision, and scalability. Static race detectors face two primary challenges: simultaneously approximating multiple conditions and inferring the correlation between locks and the memory locations they guard. We next elaborate upon each of these challenges.

Unlike the specification of most program analysis problems, the specification of a race involves several conditions. For instance, a pair of statements in a Java program is race-free if any of the following conditions is satisfied:

- The statements never access the same memory location.

- The memory location accessed by either (or both) of the statements is always thread-local (as opposed to thread-shared).
- The statements are ordered by the thread structure of the program.
- The statements are ordered by lock-based synchronization.

Each of these conditions is itself undecidable and the separate literature in static analysis for some of them is vast. Since different pairs of statements in a given program may be race-free due to different conditions, an effective static race detection algorithm must approximate all four conditions precisely. While imprecision is typically tolerable when it arises out of a reasonable static analysis for one of these conditions in isolation—the case in most static analysis clients—it can become unbearable for race detection on large, real-world programs, even when reasonable static analyses are used for each of them.

Determining whether a pair of statements is ordered by lock-based synchronization is a particularly difficult problem: it requires inferring the correlation between locks and the memory locations they guard. This problem is easy in the case of programs with coarse-grained parallelism which use global, uniquely-named locks but quickly becomes intractable in the case of programs with fine-grained parallelism which create multiple locks at run-time that are stored in data structures and passed around by functions. Indeed, folk wisdom that static race detection is intractable is primarily attributed to this problem.

1.3 Our Approach

This thesis presents an effective approach to static race detection for Java. Our algorithm consists of four static analyses, each of which conservatively approximates a different condition sufficient for proving a given pair of statements race-free:

1. A pair of statements is race-free if the statements never access the same memory location. We employ a *may alias analysis* for approximating this condition.

2. A pair of statements is race-free if the memory location accessed by either (or both) of the statements is always thread-local (as opposed to thread-shared). We employ a *thread escape analysis* for approximating this condition.
3. A pair of statements is race-free if the statements are ordered by the thread structure of the program. We employ a *may-happen-in-parallel analysis* for approximating this condition.
4. A pair of statements is race-free if the statements are ordered by lock-based synchronization. We employ *conditional must not alias analysis* for approximating this condition.

Each pair of statements in the given Java program that fails to satisfy the above four conditions is output as a potential race.

We next elaborate upon the key aspects of our algorithm by means of the following pseudo-code example:

<pre>// Thread t1 executes: synchronized (l1) { s1: e1.f = ...; }</pre>	<pre>// Thread t2 executes: synchronized (l2) { s2: e2.f = ...; }</pre>
--	--

Here, “`synchronized (l) { s }`” is Java’s lexically-scoped locking construct: the thread executing it acquires a lock on the object denoted by `l` before executing `s` and releases the lock upon finishing executing `s`. The statements labeled `s1` and `s2` write to memory locations denoted by instance field `f` of the objects denoted by expressions `e1` and `e2`, respectively. By condition (1) above, `s1` and `s2` cannot be involved in a race if `e1.f` and `e2.f` do not denote the same memory location in any execution, which in turn is true if `e1` and `e2` do not denote the same object in any execution since fields of different objects in Java have different memory locations. This condition can be approximated using a *may alias analysis* (also called *points-to analysis* or *pointer analysis*), and is denoted by the predicate $\neg \text{MAY-ALIAS}(e1, e2)$ in the literature.

The precision of the may alias analysis is key to the precision of our race detection algorithm. The other three analyses used in our algorithm to approximate conditions

(2)–(4) above, namely, thread escape analysis, may-happen-in-parallel analysis, and conditional must not alias analysis, also use the points-to information and the call graph computed by the may alias analysis. As a result, the precision of the may alias analysis is not only important in its own right, but it also affects the precision of the other analyses.

We employ a relatively recent form of may alias analysis for Java called *k-object-sensitive analysis* introduced by Milanova et al. [60, 61] that provides the precision necessary for our race detection approach. It is well known that *k-object-sensitive analysis* is difficult to scale. We use two key insights to obtain a reasonably scalable *k-object-sensitive analysis*. First, we leverage recent work on BDD-based techniques for scaling context sensitive, whole-program static analyses [8, 51, 88, 94]. Secondly, and more importantly, we employ a novel demand-driven approach to static race detection that enables an *adaptive k-object-sensitive analysis* capable of analyzing different parts of the same program with different degrees of precision. The approach achieves scalability by guiding the analysis to use high precision (i.e., *k* values bigger than one) for only a tiny fraction of (object allocation sites in) the program and low precision (i.e., *k* values equal to one) for the vast majority of (object allocation sites in) the program.

Returning to our running example, if $\neg \text{MAY-ALIAS}(\mathbf{e1}, \mathbf{e2})$ holds, then **s1** and **s2** are race-free by condition (1) above. Otherwise, we can try to prove that **s1** and **s2** are race-free using one of conditions (2)–(4). In particular, we can try to prove that **l1** and **l2** denote the same lock object in *every* execution, in which case **s1** and **s2** are race-free by condition (4). This condition can be approximated using a *must alias analysis*, and is denoted by the predicate **MUST-ALIAS**(**l1**, **l2**) in the literature.

Must-alias analysis is perceived as a harder problem than may alias analysis, and the literature on must alias analysis, unlike that on may alias analysis, is very small. Hence, the apparent need for a must alias analysis in checking whether a pair of statements is ordered by lock-based synchronization has been a major impediment to many previous static race detection approaches. Our key insight is that must alias analysis is not necessary for static race detection, and that a new analysis we call *conditional must not alias analysis* suffices. The idea behind conditional must not

alias analysis is that, instead of proving that `l1` and `l2` denote the same object in every execution, we can prove that whenever `l1` and `l2` denote different objects in an execution, `e1` and `e2` also denote different objects in that same execution. If this holds, then it is easy to see that `s1` and `s2` are race-free. Our conditional must not alias analysis is based on a novel form of object reachability analysis called *disjoint reachability analysis* that reasons about reachability in the heap between lock objects like `l1` and `l2` and accessed objects like `e1` and `e2`.

We have implemented our static race detection algorithm in a tool *Chord* and applied it to a suite of eight multithreaded Java programs. The suite includes a mix of *open programs* (incomplete programs such as libraries), *closed programs* (complete programs with a main method), programs studied in previous work on race detection, and newer, real-world, open-source programs. Modulo certain sources of unsoundness, namely, dynamic class loading, reflection, unsound modeling of missing caller code (e.g., main methods in the case of libraries), unsound modeling of missing callee code (e.g., native methods), and any remaining bugs in our implementation, our approach identifies all races in each of these programs, has a false positive rate of around 25%, and checks each of these programs in under a few minutes, taking 132 minutes to analyze our largest benchmark Apache Derby, a popular relational database engine comprising over 700K lines of Java bytecode. Finally, our approach is effective, finding tens to hundreds of previously unknown concurrency bugs in mature and widely used programs in our benchmark suite, including 1018 distinct harmful races that exposed 319 distinct bugs in Apache Derby. The usefulness of our approach is attested by the fact that many of these bugs were fixed by the programs’ developers upon reporting.

1.4 Summary of Contributions

This thesis makes the following contributions:

- It presents a novel algorithm for static race detection in Java programs. The algorithm employs four static analyses, namely, a *k*-object-sensitive may alias analysis, a thread escape analysis, a may-happen-in-parallel analysis, and a

conditional must not alias analysis. Each of these analyses conservatively approximates a different condition sufficient for proving a given pair of statements race-free while together enabling our algorithm to output a useful set of potential races.

- It introduces conditional must not alias analysis, a novel technique for correlating locks with the memory locations they guard. It circumvents the need for must alias analysis which has been a major impediment to many previous static race detection approaches. The analysis is based on a novel form of object reachability analysis called *disjoint reachability analysis* that reasons about reachability in the heap between locks and the memory locations they guard. Disjoint reachability analysis is essentially a lightweight *shape analysis* [90] and may have applications beyond static race detection.
- It presents a novel demand-driven approach to static race detection that enables an adaptive k -object-sensitive analysis capable of using different k values for different object allocation sites in the same program. The approach guides the analysis to use bigger k values for sites where it deems higher precision is necessary and lower k values for sites where it deems lower precision suffices. In practice, it uses bigger k values for few sites and smaller k values for the vast majority of sites, thereby striking a good trade-off between precision and scalability while retaining soundness.
- It describes the implementation of our static race detection algorithm in a tool named **Chord** and reports upon our experience applying it to a suite of eight diverse multithreaded Java programs. The effectiveness of our approach is validated by the discovery of tens to hundreds of previously unknown concurrency bugs in mature and widely used programs in our benchmark suite.

1.5 Organization

The rest of this thesis is organized as follows. Chapter 2 presents our basic race detection algorithm with a focus on precision. It sacrifices soundness by employing an

unsound lockset analysis for checking whether a pair of statements is ordered by lock-based synchronization, and it sacrifices scalability by using a non-adaptive k -object-sensitive analysis that uses the same k value for all object allocation sites in the given program. Chapter 3 presents conditional must not alias analysis which replaces the lockset analysis and renders our race detection algorithm sound. Chapter 4 shows how to make our race detection algorithm demand-driven, which improves its scalability by enabling an adaptive k -object-sensitive analysis capable of using different k values for different object allocation sites in the given program. Finally, Chapter 5 concludes with directions for future work.

Chapter 2

Basic Race Detection Algorithm

This chapter presents our basic race detection algorithm with a focus on precision. The algorithm consists of four static analyses—a may alias analysis, a thread escape analysis, a may-happen-in-parallel analysis, and a lockset analysis—each of which approximates a separate condition in the specification of a race while together enabling the algorithm to output a useful set of potential races. We have implemented the algorithm in a tool **Chord** and report upon our experience applying it to a suite of eight multithreaded Java programs.

2.1 Introduction

A *race* is a condition in a shared-memory multithreaded program in which a pair of threads may access the same memory location without any ordering enforced between the accesses, and at least one of the accesses is a write. Races often imply violations of program invariants. However, races are typically triggered under very specific thread schedules, and the inherent non-determinism of thread schedules renders races not only more likely to elude detection by prevalent industrial testing techniques, but also more difficult to reproduce and fix once they have been detected. As a result, race detection tools are valuable for improving the reliability of multithreaded programs.

The large body of work on race detection, discussed in detail in Section 2.12, may be broadly classified as dynamic or static. Briefly, dynamic race detectors are based

on either the *happens-before* relation [1, 19, 20, 22, 59, 71, 75], the *lockset* algorithm [2, 14, 17, 64, 74, 83, 84], or a *hybrid* approach that combines the happens-before and lockset approaches [23, 42, 65, 68, 92], while static race detectors are either primarily flow insensitive type-based systems [11, 12, 28, 29, 69, 73], flow sensitive static versions of the lockset algorithm [18, 26, 79], or path sensitive model checkers [45, 70, 76].

Static race detectors typically sacrifice some combination of soundness, precision, and scalability. The difficulty of effective static race detection is underscored by the fact that race detection tools are predominantly dynamic. State-of-the-art dynamic race detectors enjoy both precision and scalability. Like any dynamic analysis, however, they are inherently unsound and explore only a fraction of the space of the program’s inputs and thread schedules. Furthermore, they cannot be applied to open programs such as libraries, since such programs cannot be executed.

In this chapter, we present our basic approach to static race detection for Java. We dissect the specification of a race to identify four natural conditions, each of which is sufficient for proving a given pair of statements race-free, but all of which are necessary in practice since different pairs of statements in a given Java program may be race-free because of different conditions. Our race detection algorithm consists of four static analyses each of which checks a separate condition while together enabling the algorithm to report a useful set of potential races:

1. A pair of statements is race-free if they never access the same memory location. We use a *may alias analysis* to approximate this condition.
2. A pair of statements is race-free if the memory location accessed by either (or both) of the statements is always thread-local (as opposed to thread-shared). We use a *thread escape analysis* to approximate this condition.
3. A pair of statements is race-free if they are ordered by the thread structure of the program. We use a *may-happen-in-parallel analysis* to approximate this condition.

4. A pair of statements is race-free if they are ordered by lock-based synchronization. We use a *lockset analysis* to approximate this condition.

Each pair of statements in the given Java program that fails to satisfy the above four conditions is output as a potential race for manual inspection.

We have implemented our static race detection algorithm in a tool **Chord** and applied it to a suite of eight multithreaded Java programs, many of which are mature and widely used. Our approach found 406 previously unknown concurrency bugs in these programs, many of which were fixed by the programs' developers upon reporting. In Apache Derby, an open-source relational database engine, **Chord** analyzed over 700K lines of Java bytecode and reported races revealing 319 distinct bugs. Tens of bug reports in two other open-source programs, JdbF (an object-relational mapping system) and jTDS (a JDBC driver), led the developers of those programs to overhaul their synchronization. In Apache Commons Pool, an open-source generic object-pooling library that enables optimizing usage of resources like threads, sockets, database connections, etc., **Chord** exposed 17 bugs, all of which were fixed in five immediate dedicated patches.

The rest of this chapter is organized as follows. Section 2.2 illustrates our approach by means of an example Java program. Section 2.3 presents a preprocessing stage which primarily constructs a context insensitive call graph of the given Java program. Sections 2.4–2.7 present our may alias analysis, our thread escape analysis, our may-happen-in-parallel analysis, and our lockset analysis, respectively. Section 2.8 shows how to put all four analyses together to compute the set of potential races to be reported. Section 2.9 discusses usability aspects of our algorithm, namely, how it generates counterexamples to explain the detected races and how it checks open programs such as libraries. Section 2.10 discusses unsoundness issues in our algorithm. Section 2.11 describes our implementation of the algorithm in **Chord** and the results of applying it to a suite of eight multithreaded Java programs. Finally, Section 2.12 discusses related work.

2.2 Example

In this section, we present a multithreaded Java program that we use as the running example to illustrate our approach. The program, shown in Figure 2.1, begins by executing the `main` method of class `T` in an implicit main thread. The main thread first creates two integers, each encapsulated in a separate `A` object, each of which in turn is encapsulated in a separate `B` object. The main thread then creates a bunch of `T` objects in a loop. Whenever a `T` object is created in each iteration of the loop, the two `B` objects are assigned to instance fields `f1` and `f2` of that `T` object, and the `start` method of superclass `java.lang.Thread` is called on that `T` object. The `start` method, which is not shown, invokes the `run` method of class `T` on that `T` object in a fresh child thread. The calls to the `start` method are asynchronous: the main thread continues executing the `main` method while each of the previously spawned child threads executes the `run` method.

The main thread and the child threads communicate via the two integers created upfront by the main thread. All accesses to one of the two integers are consistently protected by a lock. In particular, the main thread and the child threads hold a lock on the same `B` object while accessing the corresponding encapsulated integer: the main thread holds the lock while writing to it and each child thread holds the lock while reading it. As a result, there are no races on this integer. The other integer, however, is accessed by all threads without holding any lock: the main thread repeatedly reads the integer in a loop until it becomes non-zero, and each child thread writes to the integer upon finishing executing the `run` method. As a result, there are races on this integer. This pattern, called *asynchronous notification*, is commonly used in real-world multithreaded Java programs. The race is seemingly benign but it is in fact harmful: in the absence of consistent lock-based synchronization or a `volatile` declaration of the integer, Java’s memory model [55] allows accesses to the integer to be aggressively optimized, for instance, allowing the unprotected writes by the child threads to a location in the register or cache of one processor on a multiprocessor that is never flushed, thereby preventing the writes from ever becoming visible to the main thread executing on a different processor and making it loop forever.


```

public class T extends java.lang.Thread {
    public static void main(String[] a) {
h1,i1:      B v1 = new B();
h2,i2:      B v2 = new B();
            for (int i = 0; i < *; i++) {
h3,i3:      T v3 = new T(v1, v2);
i4:         v3.start();
            }
i5:         while (v1.get() == 0) {
l1:         synchronized (v2) {
i6:         v2.set(...);
            }
        }
        private B f1, f2;
        public T(B v4, B v5) {
e1:         this.f1 = v4;
e2:         this.f2 = v5;
        }
        public void run() {
e3:         B v6 = this.f1;
e4:         B v7 = this.f2;
l2:         synchronized (v7) {
i7:         ... = v7.get();
        }
i8:         v6.set(1);
    }
}

public class B {
    private A f3;
    public B() {
h4,i9:      A v8 = new A();
e5:         this.f3 = v8;
    }
    public int get() {
e6:         A v9 = this.f3;
i10:        return v9.get();
    }
    public void set(int i) {
e7:         A v10 = this.f3;
i11:        v10.set(i);
    }
}

public class A {
    private int f4;
    public A() {
e8:         this.f4 = 0;
    }
    public int get() {
e9:         return this.f4;
    }
    public void set(int i) {
e10:        this.f4 = i;
    }
}

```

Figure 2.1: Example multithreaded Java program.

2.3 Preprocessing

Our race detection algorithm is whole-program and requires a call graph of the given Java program. Hence, the first step in our algorithm is to obtain a context insensitive call graph of the program from a user-specified main class containing the distinguished main method and a user-specified class path specifying the location of all reachable classes. (We show how our algorithm handles open programs, which typically lack a main method, in Section 2.9.2.) We obtain this call graph using Spark [52], a 0-CFA-based may alias analysis with on-the-fly call graph construction provided in the Soot compiler framework [81]. We next define some important program domains we extract from this call graph:

- $\mathbb{M}$ is the set of all reachable methods in the call graph. We use m_{main} to denote the element in $\mathbb{M}$ representing the distinguished main method, which is also the root method of the implicitly spawned main thread, and we use m_{start} to denote the element in $\mathbb{M}$ representing the `start` method in class `java.lang.Thread`, which is the root method of each explicitly spawned thread. For our running example from Figure 2.1, $\mathbb{M}$ contains `T.main` as m_{main} , `T.run`, `B.get`, `B.set`, `A.get`, `A.set`, constructor and class initializer methods of application classes `T`, `B`, and `A`, as well as methods of library classes such as `java.lang.Thread` and `java.lang.Object` which are not shown. For the remaining domains as well, we focus only on elements from application classes, although each domain contains additional elements from library classes.
- $\mathbb{I}$ contains each method invocation site in the body of each method in $\mathbb{M}$. For our running example, $\mathbb{I}$ includes statements labeled `i1` through `i11`.
- $\mathbb{H}$ contains each object allocation site in the body of each method in $\mathbb{M}$. For our running example, $\mathbb{H}$ includes statements labeled `h1` through `h4`.
- $\mathbb{V}$ contains each local variable declared in each method in $\mathbb{M}$. For our running example, $\mathbb{V}$ includes local variables `v1` through `v10`, and `this`. In our implementation, however, $\mathbb{V}$ has a separate element for each `this` variable in methods `T.run`, `B.get`, `B.set`, `A.get`, `A.set`, `T.<init>`, `B.<init>`, and `A.<init>`. We

use a single `this` variable in our presentation for brevity; the context always makes it clear which `this` variable is being referenced.

- $\mathbb{E}$ contains each statement in the body of each method in $\mathbb{M}$ that reads or writes an instance field, an array element, or a static field. For our running example, $\mathbb{E}$ includes statements labeled `e1` through `e10`.
- $\mathbb{F}$ contains each instance field read or written in the body of any method in $\mathbb{M}$, plus a hypothetical field f_{elems} that is regarded as read/written whenever an array element is read/written; this field is necessary since we do not distinguish between different elements of the same array. For our running example, $\mathbb{F}$ includes fields `f1` and `f2` of class `T`, field `f3` of class `B`, and field `f4` of class `A`.
- $\mathbb{G}$ contains each static field read or written in the body of any method in $\mathbb{M}$. Our running example does not contain any static fields in application classes `T`, `B`, and `A`.
- $\mathbb{P}$ contains each program point in the body of each method in $\mathbb{M}$. This includes the program points of object allocation sites, method invocation sites, and statements accessing instance fields, array elements, and static fields.

2.4 *k*-Object-Sensitive Analysis

The points-to information and call graph obtained using a 0-CFA-based analysis is too imprecise for our race detection approach. Specifically, the points-to information is context and object insensitive while the call graph is context insensitive.

The literature on may alias analysis is particularly extensive [46]. We experimented with a variety of may alias analyses ranging from 0-CFA-based analysis to Whaley and Lam’s *k*-CFA-based analysis [88] where *k* is the depth of the program’s call graph obtained after collapsing strongly connected components to single nodes. We eventually chose a relatively recent form of may alias analysis for Java called *k-object-sensitive analysis* introduced by Milanova et al. [60, 61] that provides the precision necessary for our race detection approach.

It is well known that k -object-sensitive analysis is difficult to scale. While k -CFA is context sensitive and the number of abstract method contexts for a given program grows exponentially with k , it is object insensitive in that the number of abstract objects is independent of k . (The object insensitivity of k -CFA is the primary reason it does not offer the precision necessary for our race detection approach.) In contrast, k -object-sensitive analysis is not only context sensitive but also object sensitive, and the number of abstract objects for a given program also grows exponentially with k (in fact, as we shall see shortly, the set of abstract objects is the same as the set of abstract method contexts). We use two key insights to obtain a reasonably scalable k -object-sensitive analysis, namely, a Binary Decision Diagram (BDD) based implementation of the analysis and a demand-driven approach to static race detection. In this chapter, however, we focus only on the insight concerning the BDD-based implementation; Chapter 4 further improves upon scalability by making the race detection algorithm presented in this chapter demand-driven.

Recent work has demonstrated the effectiveness of BDD-based techniques in scaling context sensitive, whole-program static analyses [8, 51, 88, 94]. Our implementation of k -object-sensitive analysis leverages these advances, in particular, we use `bddbddb` [48, 87] to express the analysis declaratively using Datalog constraints over program relations. A BDD is a graph-based data structure for representing and manipulating a boolean relation [13]. BDDs are particularly effective at compacting representing and efficiently manipulating relations with high levels of redundancy, such as those arising in context sensitive, whole program static analyses. Hence, all relations used in our analysis, including input relations that encode basic program facts and output relations that encode the points-to information and call graph, are represented as BDDs, while the Datalog constraints over these relations are implemented as operations on BDDs. The memory consumption of BDDs and the running time of BDD operations depends upon how effectively the redundancy in the represented relations is exploited, which in turn depends heavily on a client-specified ordering of the domains of those relations. In our analysis, an ordering of the various program domains (e.g., $\mathbb{M}$, $\mathbb{I}$, $\mathbb{H}$, etc. from Section 2.3) carefully chosen once and for all enables the redundancy in program relations to be exploited effectively, reducing

the memory consumption of BDDs and the running time of BDD operations and thereby enabling the analysis to scale.

Our implementation of k -object-sensitive analysis is parameterized by a positive integer k that may be instantiated differently for different programs. We call this variant of k -object-sensitive analysis *non-adaptive*. In Chapter 4, we present an *adaptive* variant that allows different k values to be used for different object allocation sites in the same program, which enables a demand-driven race detection algorithm that strikes a good trade-off between scalability and precision by using bigger k values for a few sites and smaller k values for the vast majority of sites in the program. In this chapter, however, we presume a *non-adaptive* k -object-sensitive analysis.

The analysis is *object sensitive*, that is, it abstracts different objects allocated at the same site by potentially different abstract objects. An abstract object o is a non-empty sequence of at most k object allocation sites, denoted $[h_n :: \dots :: h_1]$. We call h_n and h_1 the most and least significant sites, respectively. Suppose $o.\mathbf{car}$ denotes the head h_n and $o.\mathbf{cdr}$ denotes the tail $[h_{n-1} :: \dots :: h_1]$. Then, different objects allocated at site h_n are abstracted by different abstract objects o_1 and o_2 iff $o_1.\mathbf{car} = o_2.\mathbf{car} = h_n$ and $o_1.\mathbf{cdr} \neq o_2.\mathbf{cdr}$. We use $\mathbb{O}$ to denote the set of all abstract objects.

The analysis is *context sensitive*, that is, it analyzes each method in potentially multiple abstract contexts. An abstract context is either a distinguished context ϵ which denotes the sole context in which the main method m_{main} is analyzed, or it is an abstract object. We use $\mathbb{C}$ to denote the set of all abstract contexts.

The analysis is flow insensitive. The lack of flow sensitivity, however, does not adversely affect the precision of the analysis because our implementation operates on an SSA-like representation of the given Java program.

The analysis outputs the following relations:

- $\mathbf{ptsV} : \mathbb{C} \times \mathbb{V} \times \mathbb{O}$, the points-to information for local variables, contains each tuple (c, v, o) such that local variable v may point to abstract object o in abstract context c of v 's declaring method. Note that the points-to information is both context and object sensitive.
- $\mathbf{ptsG} : \mathbb{G} \times \mathbb{O}$, the points-to information for static fields, contains each tuple

(g, o) such that static field g may point to abstract object o . Note that the points-to information is object sensitive but not context sensitive: static fields in Java are akin to global variables that are declared outside of all methods, and abstract contexts are associated only with methods.

- **heap** : $\mathbb{O} \times \mathbb{F} \times \mathbb{O}$, the heap abstraction, contains each tuple (o_1, f, o_2) such that instance field f or the hypothetical field f_{elems} of abstract object o_1 may point to abstract object o_2 .
- **cscg** : $\mathbb{C} \times \mathbb{I} \times \mathbb{C} \times \mathbb{M}$, the call graph, contains each tuple (c_1, i, c_2, m) such that method invocation site i in abstract context c_1 of its containing method may call method m in abstract context c_2 . Note that the call graph is context sensitive.

The above relations computed for our running example from Figure 2.1 using k -object-sensitive analysis instantiated with $k = 2$ are shown in Figure 2.2. The analysis begins by regarding the main method **T.main** (as well as each class initializer method in $\mathbb{M}$) as reachable in abstract context ϵ . Whenever a method is deemed reachable in a particular abstract context, so are all statements in the body of that method. Of particular interest are object allocation sites, accesses to instance fields, accesses to array elements, accesses to static fields, and method invocation sites; we next discuss each of these cases in turn.

Object Allocation Sites

Suppose a method deemed reachable in abstract context c contains an object allocation site labeled h of the form $v = \text{new } \dots$ where v is a local variable of the method. Then, the analysis adds tuple (c, v, o) to relation **ptsV**, where o is determined as follows:

- If c is of the form ϵ , then $o \equiv [h]$. For instance, in our example, the tuple $(\epsilon, \text{v1}, [\text{h1}])$ is added to **ptsV** because site **h1** is contained in method **T.main** which is deemed reachable in abstract context ϵ .

<pre> cscg : $\mathbb{C} \times \mathbb{I} \times \mathbb{C} \times \mathbb{M} = \{$ (ϵ, i1, [h1], B.<init>) (ϵ, i2, [h2], B.<init>) (ϵ, i3, [h3], T.<init>) (ϵ, i4, [h3], T.run) (ϵ, i5, [h1], B.get) (ϵ, i6, [h2], B.set) ([h3], i7, [h2], B.get) ([h3], i8, [h1], B.set) ([h1], i9, [h4::h1], A.<init>) ([h2], i9, [h4::h2], A.<init>) ([h1], i10, [h4::h1], A.get) ([h2], i10, [h4::h2], A.get) ([h1], i11, [h4::h1], A.set) ([h2], i11, [h4::h2], A.set) $\}$ heap : $\mathbb{C} \times \mathbb{F} \times \mathbb{C} = \{$ ([h3], f1, [h1]) ([h3], f2, [h2]) ([h1], f3, [h4::h1]) ([h2], f3, [h4::h2]) $\}$ </pre>	<pre> ptsV : $\mathbb{C} \times \mathbb{V} \times \mathbb{C} = \{$ (ϵ, v1, [h1]) (ϵ, v2, [h2]) (ϵ, v3, [h3]) ([h3], v4, [h1]) ([h3], v5, [h2]) ([h3], v6, [h1]) ([h3], v7, [h2]) ([h1], v8, [h4::h1]) ([h2], v8, [h4::h2]) ([h1], v9, [h4::h1]) ([h2], v9, [h4::h2]) ([h1], v10, [h4::h1]) ([h2], v10, [h4::h2]) ([h1], this, [h1]) ([h2], this, [h2]) ([h3], this, [h3]) ([h4::h1], this, [h4::h1]) ([h4::h2], this, [h4::h2]) $\}$ ptsG : $\mathbb{F} \times \mathbb{C} = \{$ $\}$ </pre>
--	--

Figure 2.2: k -object-sensitive analysis of example program using $k = 2$.

- If c is of the form $[h_n :: \dots :: h_1]$ and $n < k$, then $o \equiv [h :: h_n :: \dots :: h_1]$. For instance, in our example, the tuple $([h1], v8, [h4::h1])$ is added to **ptsV** because site **h4** is contained in method **B.<init>** which is deemed reachable in abstract context $[h1]$, and also because we have presumed that the analysis is instantiated with $k = 2$.
- If c is of the form $[h_n :: \dots :: h_1]$ and $n \geq k$, then $o \equiv [h :: h_n :: \dots :: h_2]$, that is, the least significant site h_1 is dropped. Consider the tuple $([h1], v8, [h4::h1])$ from the previous item. If we had instantiated the analysis with $k = 1$ instead of $k = 2$, then the analysis would infer the abstract object pointed to by **v8** in abstract context $[h1]$ as $[h4]$ instead of $[h4::h1]$. This example also illustrates why lower k values may result in imprecision: if the analysis is instantiated using $k = 1$, the computed points-to information makes fewer distinctions, namely, abstract objects $[h4::h1]$ and $[h4::h2]$ in each tuple of relation **ptsV** shown in Figure 2.2 are replaced by abstract object $[h4]$.

Reads of Instance Fields or Array Elements

Suppose a method deemed reachable in abstract context c contains a statement of the form $v_2 = v_1.f$ where v_1 and v_2 are local variables of the method and f is either an instance field or the hypothetical field f_{elems} . Then, for each $(c, v_1, o_1) \in \mathbf{ptsV}$ and for each $(o_1, f, o_2) \in \mathbf{heap}$, the analysis adds tuple (c, v_2, o_2) to **ptsV**. For instance, in our example, the tuple $([h3], v6, [h1])$ gets added to **ptsV** because the heap read **v6 = this.f1** is contained in method **T.run** which is deemed reachable in abstract context $[h3]$, and because **ptsV** contains tuple $([h3], \mathbf{this}, [h3])$ and **heap** contains tuple $([h3], f1, [h1])$.

Writes to Instance Fields or Array Elements

Suppose a method deemed reachable in abstract context c contains a statement of the form $v_1.f = v_2$ where v_1 and v_2 are local variables of the method and f is either an instance field or the hypothetical field f_{elems} . Then, for each $(c, v_1, o_1) \in \mathbf{ptsV}$ and for each $(c, v_2, o_2) \in \mathbf{ptsV}$, the analysis adds tuple (o_1, f, o_2) to **heap**. For instance, in

our example, the tuple $([h1], f3, [h4::h1])$ gets added to **heap** because the heap write `this.f3 = v8` is contained in method `B.<init>` which is deemed reachable in abstract context $[h1]$, and because **ptsV** contains tuples $([h1], \text{this}, [h1])$ and $([h1], v8, [h4::h1])$.

Reads of Static Fields

Suppose a method deemed reachable in abstract context c contains a statement of the form $v = g$ where g is a static field and v is a local variable. Then, for each $(g, o) \in \text{ptsG}$, we add (c, v, o) to **ptsV**.

Writes to Static Fields

Suppose a method deemed reachable in abstract context c contains a statement of the form $g = v$ where g is a static field and v is a local variable. Then, for each $(c, v, o) \in \text{ptsV}$, we add (g, o) to **ptsG**.

Method Invocation Sites

Suppose a method deemed reachable in abstract context c contains a method invocation site i . There are two cases depending upon the kind of i :

- If i is an **invokestatic** call (a statically-dispatched call to a static method), suppose its target is static method m . Then, the analysis deems method m reachable in context c , and adds tuple (c, i, c, m) to relation **cscg**.
- If i is an **invokespecial** call (a statically-dispatched call to an instance method) or an **invokevirtual** or **invokeinterface** call (a dynamically-dispatched call to an instance method), then the points-to information is consulted. Specifically, suppose the distinguished 0th **this** argument of i is local variable v . Then, for every abstract object o pointed to by v in abstract context c , that is, for every $(c, v, o) \in \text{ptsV}$, the analysis deems method m reachable in abstract context o , and adds tuple (c, i, o, m) to relation **cscg**, where m is determined solely by i if it is an **invokespecial** call, or by both i and o using class hierarchy analysis if

it is an `invokevirtual` or `invokeinterface` call (recall that `invokespecial` is a statically-dispatched call and hence the abstract object o is not necessary to determine the target method).

In our example, `i1`, `i2`, `i3`, and `i9` are `invokespecial` calls and the rest are `invokevirtual` calls. Consider, for instance, call site `i10`. It is contained in method `B.get`, which is deemed reachable in abstract contexts `[h1]` and `[h2]`. The 0^{th} argument of the call site, `v9`, may point to abstract objects `[h4::h1]` and `[h4::h2]` in abstract contexts `[h1]` and `[h2]`, respectively, as denoted by the corresponding tuples in `ptsV`. Hence, tuples $([h1], i10, [h4::h1], A.get)$ and $([h2], i10, [h4::h2], A.get)$ are added to `cscg`.

2.4.1 Computation of originalRaces

The call graph obtained from k -object-sensitive analysis is used to compute an initial over-approximation of the set of races in the given Java program. This computation is specified as Algorithm 2.1 using `bddbldb` notation [48, 87], which consists of three sections: the `DOMAINS` section declares domains, the `RELATIONS` section declares relations over the declared domains, and also specifies whether each relation is an input relation, an output relation, or neither (in which case it is an intermediate relation), and the `RULES` section specifies the computation itself as Datalog constraints over the declared relations.

Besides the context sensitive call graph relation `cscg` obtained from k -object-sensitive analysis, the algorithm requires the following basic program relations:

- **EF** : $\mathbb{E} \times \mathbb{F}$ contains each tuple (e, f) such that statement e accesses instance field f or an array element (in which case f is the hypothetical field f_{elems}).
- **EG** : $\mathbb{E} \times \mathbb{G}$ contains each tuple (e, g) such that statement e accesses static field g .
- **ME** : $\mathbb{M} \times \mathbb{E}$ contains each tuple (m, e) such that statement e is contained in method m .

- $\mathbf{wr} : \mathbb{E}$ contains each statement e that is a write (as opposed to a read) of an instance field, a static field, or an array element.

Algorithm 2.1. Computation of `originalRaces`.

DOMAINS

- $\mathbb{C}$ abstract contexts
- $\mathbb{E}$ statements accessing a field or array element
- $\mathbb{F}$ instance fields
- $\mathbb{G}$ static fields
- $\mathbb{I}$ method invocation sites
- $\mathbb{M}$ methods

RELATIONS

- input $\mathbf{cscg} : \mathbb{C} \times \mathbb{I} \times \mathbb{C} \times \mathbb{M}$
- input $\mathbf{EF} : \mathbb{E} \times \mathbb{F}$
- input $\mathbf{EG} : \mathbb{E} \times \mathbb{G}$
- input $\mathbf{ME} : \mathbb{M} \times \mathbb{E}$
- input $\mathbf{wr} : \mathbb{E}$
- $\mathbf{reaches} : \mathbb{C} \times \mathbb{C} \times \mathbb{M}$
- output $\mathbf{originalRaces} : \mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

RULES

$$\mathbf{reaches}(\epsilon, \epsilon, m_{main}). \quad (2.1)$$

$$\mathbf{reaches}(c, c, m) \text{ :- } \mathbf{reaches}(_, c', m'), \mathbf{MI}(m', i), \mathbf{cscg}(c', i, c, m_{start}). \quad (2.2)$$

$$\begin{aligned} \mathbf{reaches}(t, c, m) \text{ :- } \mathbf{reaches}(t, c', m'), \mathbf{MI}(m', i), \mathbf{cscg}(c', i, c, m), \\ m \neq m_{start}. \end{aligned} \quad (2.3)$$

$$\begin{aligned} \mathbf{originalRaces}(t, c, e, t', c', e') \text{ :- } \mathbf{reaches}(t, c, m), \mathbf{reaches}(t', c', m'), \\ \mathbf{ME}(m, e), \mathbf{ME}(m', e'), \mathbf{EF}(e, f), \mathbf{EF}(e', f), \mathbf{wr}(e), e < e'. \end{aligned} \quad (2.4)$$

$$\begin{aligned} \text{originalRaces}(t, c, e, t', c', e') &:- \text{reaches}(t, c, m), \text{reaches}(t', c', m'), \\ &\text{ME}(m, e), \text{ME}(m', e'), \text{EF}(e, f), \text{EF}(e', f), \text{wr}(e'), e < e'. \end{aligned} \quad (2.5)$$

$$\begin{aligned} \text{originalRaces}(t, c, e, t', c', e') &:- \text{reaches}(t, c, m), \text{reaches}(t', c', m'), \\ &\text{ME}(m, e), \text{ME}(m', e'), \text{EG}(e, g), \text{EG}(e', g), \text{wr}(e), e < e'. \end{aligned} \quad (2.6)$$

$$\begin{aligned} \text{originalRaces}(t, c, e, t', c', e') &:- \text{reaches}(t, c, m), \text{reaches}(t', c', m'), \\ &\text{ME}(m, e), \text{ME}(m', e'), \text{EG}(e, g), \text{EG}(e', g), \text{wr}(e'), e < e'. \end{aligned} \quad (2.7)$$

$$\begin{aligned} \text{originalRaces}(t, c, e, t', c', e) &:- \text{reaches}(t, c, m), \text{reaches}(t', c', m), \\ &\text{ME}(m, e), \text{wr}(e), c < c'. \end{aligned} \quad (2.8)$$

$$\begin{aligned} \text{originalRaces}(t, c, e, t', c, e) &:- \text{reaches}(t_1, c, m), \text{reaches}(t', c, m), \\ &\text{ME}(m, e), \text{wr}(e), t \leq t'. \end{aligned} \quad (2.9)$$

□

The intermediate relation **reaches** is computed by Rules (2.1)–(2.3). It contains each tuple (t, c, m) such that abstract thread t may execute method m in abstract context c . Specifically, t is an abstract context of the root method of the thread being abstracted, that is, it is an abstract context of either the main method m_{main} (in the case in which the thread being abstracted is the implicit main thread) or m_{start} , the **start** method of class `java.lang.Thread` (in the case in which the thread being abstracted is an explicitly spawned thread). In the former case, the abstract context is always ϵ (since this is the sole abstract context of m_{main}), and in the latter case, it is always of the form $[h_n :: \dots :: h_1]$ (since the **start** method is an instance method and therefore all its abstract contexts under k -object-sensitive analysis are abstract objects of its distinguished 0^{th} **this** argument).

Rules (2.4)–(2.9) compute **originalRaces**, our initial over-approximation of the set of races in the given Java program. It contains each tuple $(t_1, c_1, e_1, t_2, c_2, e_2)$ such that abstract threads t_1 and t_2 may execute statements e_1 and e_2 in abstract contexts c_1 and c_2 of their containing methods, respectively. Moreover, the rules ensure that:

- e_1 and e_2 both access the same instance field, or both access array elements (that is, both access the hypothetical field f_{elems}), or both access the same static field.

#	originalRaces	aliasingRaces	escapingRaces	parallelRaces	unlockedRaces	ultimateRaces
1	$(\epsilon, [h3], e1, \epsilon, [h3], e1)$	✓				
2	$(\epsilon, [h3], e1, [h3], [h3], e3)$	✓		✓	✓	
3	$(\epsilon, [h3], e2, \epsilon, [h3], e2)$	✓				
4	$(\epsilon, [h3], e2, [h3], [h3], e4)$	✓		✓	✓	
5	$(\epsilon, [h1], e5, \epsilon, [h1], e5)$	✓				
6	$(\epsilon, [h1], e5, \epsilon, [h2], e5)$					
7	$(\epsilon, [h2], e5, \epsilon, [h1], e5)$					
8	$(\epsilon, [h2], e5, \epsilon, [h2], e5)$	✓				
9	$(\epsilon, [h1], e5, \epsilon, [h1], e6)$	✓				
10	$(\epsilon, [h1], e5, [h3], [h2], e6)$				✓	
11	$(\epsilon, [h2], e5, \epsilon, [h1], e6)$					
12	$(\epsilon, [h2], e5, [h3], [h2], e6)$	✓			✓	
13	$(\epsilon, [h1], e5, \epsilon, [h2], e7)$					
14	$(\epsilon, [h1], e5, [h3], [h1], e7)$	✓			✓	
15	$(\epsilon, [h2], e5, \epsilon, [h2], e7)$	✓				
16	$(\epsilon, [h2], e5, [h3], [h1], e7)$				✓	
17	$(\epsilon, [h4::h1], e8, \epsilon, [h4::h1], e8)$	✓				
18	$(\epsilon, [h4::h1], e8, \epsilon, [h4::h2], e8)$					
19	$(\epsilon, [h4::h2], e8, \epsilon, [h4::h1], e8)$					
20	$(\epsilon, [h4::h2], e8, \epsilon, [h4::h2], e8)$	✓				
21	$(\epsilon, [h4::h1], e8, \epsilon, [h4::h1], e9)$	✓				
22	$(\epsilon, [h4::h2], e8, \epsilon, [h4::h1], e9)$					
23	$(\epsilon, [h4::h1], e8, [h3], [h4::h2], e9)$				✓	
24	$(\epsilon, [h4::h2], e8, [h3], [h4::h2], e9)$	✓			✓	
25	$(\epsilon, [h4::h1], e8, \epsilon, [h4::h2], e10)$					
26	$(\epsilon, [h4::h2], e8, \epsilon, [h4::h2], e10)$	✓				
27	$(\epsilon, [h4::h1], e8, [h3], [h4::h1], e10)$	✓			✓	
28	$(\epsilon, [h4::h2], e8, [h3], [h4::h1], e10)$				✓	
29	$(\epsilon, [h4::h2], e10, \epsilon, [h4::h2], e10)$	✓	✓			
30	$(\epsilon, [h4::h2], e10, [h3], [h4::h1], e10)$		✓	✓	✓	
31	$([h3], [h4::h1], e10, \epsilon, [h4::h2], e10)$		✓	✓	✓	
32	$([h3], [h4::h1], e10, [h3], [h4::h1], e10)$	✓	✓	✓		
33	$(\epsilon, [h4::h1], e9, \epsilon, [h4::h2], e10)$		✓			
34	$(\epsilon, [h4::h1], e9, [h3], [h4::h1], e10)$	✓	✓	✓	✓	✓
35	$([h3], [h4::h2], e9, \epsilon, [h4::h2], e10)$	✓	✓	✓		
36	$([h3], [h4::h2], e9, [h3], [h4::h1], e10)$		✓	✓		

Table 2.1: Races in example program.

Java’s semantics ensures that these are the only three cases in which a pair of statements may access the same memory location, and hence these are the only three cases in which a pair of statements may be involved in a race.

- At least one of e_1 and e_2 is a write (since there cannot be a race between two reads).
- There is no duplication, for instance, it is never the case that **originalRaces** contains both tuples $(t_1, c_1, e_1, t_2, c_2, e_2)$ and $(t_2, c_2, e_2, t_1, c_1, e_1)$.

The relation **originalRaces** computed for our running example from Figure 2.1 is shown in the leftmost column of Table 2.1. There are two abstract threads in this example: ϵ abstracting the implicit main thread and **[h3]** abstracting each explicitly spawned child thread. The first two tuples in the table are potential races on field **f1**, the next two tuples are potential races on field **f2**, tuples 5–16 are potential races on field **f3**, and tuples 17–36 are potential races on field **f4**. This distribution of exponentially increasing numbers of potential races from fields of “external” classes like **T** to those of “internal” classes like **A** is typical in real-world Java programs for two reasons. First, viewing the heap as a tree-like structure, methods of external classes like **T** directly operate only on parts of the heap closer to the root while methods of internal classes like **A** directly operate only on parts of the heap closer to the leaves, and most heap manipulation typically occurs closer to the leaves while the parts of the heap closer to the root remain relatively unchanged. Secondly, internal classes are more heavily reused than external classes, and so more abstract contexts are needed to disambiguate accesses occurring in methods of internal classes, resulting in more potential races on fields of internal classes.

Note that we have merely shown potential races for the application classes in this example; the majority of potential races in practice come from library code that is exercised by application code because when considering the entire program, all application classes may be viewed as external classes while the library classes may be viewed as internal classes.

2.4.2 Computation of `aliasingRaces`

The points-to information computed by k -object-sensitive analysis is used to approximate the first of the four conditions, namely, that a pair of statements cannot be involved in a race if they never access the same memory location. This computation is specified as Algorithm 2.2.

Algorithm 2.2. Computation of `aliasingRaces`.

DOMAINS

$\mathbb{C}$ abstract contexts

$\mathbb{E}$ statements accessing a field or array element

$\mathbb{G}$ static fields

$\mathbb{V}$ local variables

RELATIONS

input $\text{ptsV} : \mathbb{C} \times \mathbb{V} \times \mathbb{C}$

input $\text{EG} : \mathbb{E} \times \mathbb{G}$

input $\text{EV} : \mathbb{E} \times \mathbb{V}$

output $\text{aliasingRaces} : \mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

RULES

$$\begin{aligned} \text{aliasingRaces}(t_1, c_1, e_1, t_2, c_2, e_2) &:- \text{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\ &\quad \text{EG}(e_1, g), \text{EG}(e_2, g). \end{aligned} \tag{2.10}$$

$$\begin{aligned} \text{aliasingRaces}(t_1, c_1, e_1, t_2, c_2, e_2) &:- \text{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\ &\quad \text{EV}(e_1, v_1), \text{ptsV}(c_1, v_1, o), \text{EV}(e_2, v_2), \text{ptsV}(c_2, v_2, o). \end{aligned} \tag{2.11}$$

□

Besides the context and object sensitive point-to relation `ptsV` obtained from k -object-sensitive analysis, the algorithm requires basic program relations $\text{EG} : \mathbb{E} \times \mathbb{G}$ containing each tuple (e, g) such that statement e accesses static field g , and $\text{EV} : \mathbb{E} \times \mathbb{V}$

containing each tuple (e, v) such that statement e accesses an instance field or array element, and e reads local variable v to refer to the object whose instance field or array element is accessed.

Rules (2.10) and (2.11) compute **aliasingRaces**, a subset of **originalRaces** obtained by eliminating each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \mathbf{originalRaces}$ that is deemed race-free because e_1 and e_2 never access the same memory location in abstract contexts c_1 and c_2 of their respective containing methods. (Note that this computation does not inspect abstract threads t_1 and t_2 in the tuple.) Recall from the computation of **originalRaces** in Algorithm 2.1 that e_1 and e_2 either both access the same instance field or both access array elements or both access the same static field; we next consider each of these cases in turn.

Rule (2.10) considers the case in which a static field is accessed. In this case, the tuple cannot be eliminated because static fields in Java are akin to global variables, and all accesses to the same global variable refer to the same global location.

Rule (2.11) considers the case in which an instance field or an array element (that is, the hypothetical field f_{elems}) is accessed. Suppose e_1 and e_2 contain accesses of the form $v_1.f$ and $v_2.f$. The rule says that the tuple is retained if v_1 and v_2 may point to the same abstract object o in abstract contexts c_1 and c_2 , respectively. If this condition does not hold, then k -object-sensitive analysis ensures that v_1 and v_2 can never refer to the same object in abstract contexts c_1 and c_2 , and Java's semantics ensures that if v_1 and v_2 do not refer to the same object, then fields f of those objects do not have the same memory location, and hence e_1 and e_2 cannot be involved in a race and the tuple can be eliminated.

The relation **aliasingRaces** computed for our running example from Figure 2.1 is shown in the second column of Table 2.1. A $\checkmark$ mark in a particular row in that column indicates that the corresponding tuple in **originalRaces** (shown in the first column of the same row) is retained in **aliasingRaces**. For instance, consider tuples #9 and #10, both of which are potential races involving accesses **e5** and **e6**, which are of the form **this.f3 = v8** and **v9 = this.f3**. Tuple #9 is retained because it involves accesses **e5** and **e6** in the same abstract context [**h1**] of their containing methods, and the **this** variable in both accesses points to the same abstract object [**h1**] in

abstract context `[h1]`. But tuple #10 is eliminated since it involves accesses `e5` and `e6` in different abstract contexts `[h1]` and `[h2]`, respectively, and the `this` variable in accesses `e5` and `e6` points to distinct abstract objects `[h1]` and `[h2]` in abstract contexts `[h1]` and `[h2]`, respectively. In all, this computation alone eliminates 44% (16/36) of the tuples in `originalRaces` in our example.

2.5 Thread Escape Analysis

In this section, we present our thread escape analysis and its application to our race detection algorithm.

The goal of a thread escape analysis is to determine whether an object allocated by a thread may be visible to another thread. The classic application of thread escape analysis for Java is elimination of unnecessary lock-based synchronization [6, 9, 10, 15, 72, 89]. Such unnecessary synchronization occurs frequently in Java programs, for instance, the lock-based synchronization provided in methods of a thread-safe library class like `java.util.Vector` is clearly unnecessary when such methods are called from a single-threaded context, and it is redundant when such methods are called from a multithreaded context that provides its own synchronization. In our experience, however, static race detection demands a thread escape analysis of significantly higher precision than that required for lock-based synchronization elimination for two reasons. First, an imprecise thread escape analysis for lock-based synchronization elimination causes fewer lock operations to be eliminated, which in turn simply causes the program to run slower. An imprecise thread escape analysis for static race detection, on the other hand, causes fewer non-races to be eliminated, which in turn causes more false races to be reported. Manually inspecting race reports, however, is a very time-consuming task (see Section 2.9.1). Secondly, the set of objects on which locks are held is typically a small fraction of the set of objects on which races may occur and, likewise, the (static) number of lock operations is typically a small fraction of the (static) number of operations that may be involved in races. For these reasons, our thread escape analysis is context, object, and flow sensitive. All previous thread escape analyses we are aware of are context and object insensitive, though many are

flow sensitive. Our thread escape analysis is context and object sensitive in the same sense as our k -object-sensitive analysis, namely, the set of abstract contexts is $\mathbb{C}$ and the set of abstract objects is $\mathbb{O}$.

Our thread escape analysis produces the following relations:

- **ptsIn**, **ptsOut** : $\mathbb{C} \times \mathbb{P} \times \mathbb{V} \times \mathbb{O}$ represents the the points-to sets of local variables at the entry and exit, respectively, of each program point in each abstract context of its containing method. Specifically, relation **ptsIn** (resp. **ptsOut**) contains each tuple (c, p, v, o) such that local variable v may point to abstract object o at the entry (resp. exit) of program point p in abstract context c of p 's containing method.
- **heapIn**, **heapOut** : $\mathbb{C} \times \mathbb{P} \times \mathbb{O} \times \mathbb{F} \times \mathbb{O}$ represents the heap abstraction at the entry and exit, respectively, of each program point in each abstract context of its containing method. Specifically, relation **heapIn** (resp. **heapOut**) contains each tuple (c, p, o_1, f, o_2) such that field f of abstract object o_1 may point to abstract object o_2 at the entry (resp. exit) of program point p in abstract context c of p 's containing method, where f is either an instance field or the hypothetical field f_{elems} .
- **escIn**, **escOut** : $\mathbb{C} \times \mathbb{P} \times \mathbb{O}$ represents the thread-escaping sets at the entry and exit, respectively, of each program point in each abstract context of its containing method. Specifically, relation **escIn** (resp. **escOut**) contains each tuple (c, p, o) such that abstract object o may escape the current thread at the entry (resp. exit) of program point p in abstract context c of p 's containing method.

We describe the intraprocedural and interprocedural components of the analysis separately. We assume that each method has a unique entry point without any predecessors and a unique exit point without any successors.

2.5.1 Intraprocedural Analysis

Consider any program point p other than the entry point of a method in any abstract context c of that method.

The analysis computes the points-to sets of local variables, the heap abstraction, and the thread-escaping set at the entry of p in abstract context c as the union of the points-to sets of the corresponding local variables, the union of the heap abstractions, and the union of the thread-escaping sets, respectively, at the exit of each immediate predecessor of p in the same abstract context c .

The analysis computes the points-to sets of local variables, the heap abstraction, and the thread-escaping set at the exit of p in abstract context c as follows, depending upon the kind of statement at p .

Object Allocation Sites

Suppose the statement is an object allocation site labeled h of the form $v = \text{new } \dots$ where v is a local variable. Recall that our k -object-sensitive analysis computes the abstract object pointed to by v as either $[h]$ (if c is ϵ) or $[h :: h_n :: \dots :: h_1]$ (if c is of the form $[h_n :: \dots :: h_1]$ and $n < k$) or $[h :: h_n :: \dots :: h_2]$ (if c is of the form $[h_n :: \dots :: h_1]$ and $n \geq k$). Suppose this abstract object is denoted o . Then, the points-to set of v at exit is $\{o\}$. The points-to sets of local variables other than v , the heap abstraction, and the thread-escaping set at exit are the same as those at entry.

Reads of Instance Fields or Array Elements

Suppose the statement is of the form $v_2 = v_1.f$ where v_1 and v_2 are local variables and f is either an instance field or the hypothetical field f_{elems} . Then, the points-to set of each local variable other than v_2 at exit is the same as that at entry. The heap abstraction at exit is also the same as that at entry. The points-to set of v_2 at exit is determined as follows. Suppose v_1 contains abstract object o in its points-to set at entry. There are two cases:

- If o is not in the thread-escaping set at entry, then the points-to set of v_2 at exit contains each abstract object to which field f of o may point in the heap

abstraction at entry.

- If o is in the thread-escaping set at entry, however, then the points-to set of v_2 at exit contains each abstract object o' to which field f of o may point in the *global* heap abstraction represented by relation **heap** computed by our k -object-sensitive alias analysis. In this case, o' is also added to the thread-escaping set at exit.

In either case, the thread-escaping set at exit contains every abstract object in the thread-escaping set at entry.

Writes to Instance Fields or Array Elements

Suppose the statement is of the form $v_1.f = v_2$ where v_1 and v_2 are local variables and f is either an instance field or the hypothetical field f_{elems} . The points-to sets of local variables at exit are the same as those at entry. The heap abstraction at exit extends the heap abstraction at entry to capture the effect of the heap write: for each pair of abstract objects o_1 and o_2 to which v_1 and v_2 respectively may point at entry, field f of o_1 may point to o_2 in the heap abstraction at exit. The thread-escaping set at exit contains every abstract object in the thread-escaping set at entry. Furthermore, if o_1 is in the thread-escaping set at entry but o_2 is not, then each abstract object reachable from o_2 in the heap abstraction at exit is added to the thread-escaping set at exit.

Reads of Static Fields

Suppose the statement is of the form $v = g$ where g is a static field and v is a local variable. The points-to sets of local variables other than v at exit are the same as those at entry. The heap abstraction at exit is the also same as that at entry. The points-to set of v at exit is the same as the points-to set of g represented by relation **ptsG** computed by our k -object-sensitive analysis. The thread-escaping set at exit contains each abstract object in the thread-escaping set at entry, plus each abstract object reachable in the heap abstraction at exit from any abstract object in the points-to set of v at exit, as it may be reachable from any thread via g .

Writes to Static Fields

Suppose the statement is of the form $g = v$ where g is a static field and v is a local variable. The points-to sets of all local variables at exit are the same as those at entry. The heap abstraction at exit is also the same as that at entry. Finally, the thread-escaping set at exit contains each abstract object in the thread-escaping set at entry, plus each abstract object reachable in the heap abstraction at exit from any abstract object in the points-to set of v at exit, as it may be reachable from any thread via g .

2.5.2 Interprocedural Analysis

We now present the interprocedural component of our thread escape analysis. We first describe how information at the entry of a method is computed from its callers and next describe how information at the exit of a method invocation site is computed from its callees.

Consider a method m with unique entry point p in abstract context c . The points-to set of each local variable of m at the entry of p in abstract context c is the empty set if it is not a formal argument. Otherwise, it is the union of the points-to sets of the corresponding actual argument at the entry of each method invocation site i in abstract context c' such that $(c', i, c, m) \in \text{cscg}$ (recall that **cscg** represents the context-sensitive call graph of the program computed by our k -object-sensitive analysis). The treatment of the 0^{th} formal argument is different if m is an instance method (as opposed to a static method); in this case, the points-to set of the 0^{th} formal argument contains only those abstract objects in the points-to set of the 0^{th} actual argument of i that are either c or c^* . This is because k -object-sensitive analysis ensures that for other abstract objects in the points-to set of the 0^{th} actual argument of i , either the target method of i in abstract context c' is different from m or it is m in an abstract context different from c .

The heap abstraction and thread-escaping set at the entry of p in abstract context c is the union of the heap abstractions and the union of the thread-escaping sets, respectively, at the entry of each method invocation site i in abstract context c' such

that $(c', i, c, m) \in \text{cscg}$. Furthermore, if $m = m_{start}$, that is, m is the `start` method of class `java.lang.Thread`, then each abstract object reachable in the heap abstraction from any abstract object in the points-to set of the 0^{th} formal argument is added to the thread-escaping set. Note that our thread escape analysis is not summary-based. We reduce imprecision by eliminating parts of the heap abstraction and the thread-escaping set computed at the entry of p in abstract context c that are not be read by any statement in the body of m and any method called directly or transitively by m .

Next, consider a method invocation site i in abstract context c of its containing method. Suppose local variable v is a return variable of i . The points-to sets of local variables other than v at the exit of i in abstract context c are the same as those at the entry of i , while that of v is the union of the points-to sets of the return variable at the exit of the unique exit point of each method m in abstract context c' such that $(c, i, c', m) \in \text{cscg}$.

The heap abstraction and thread-escaping set at the exit of i in abstract context c is the union of the heap abstractions and the union of the thread-escaping sets, respectively, at the exit of the unique exit point of each method m in abstract context c' such that $(c, i, c', m) \in \text{cscg}$, as well as that at the entry of i in abstract context c . The latter is necessary because, as we argued above, we may eliminate parts of the heap abstraction and the thread escaping set while propagating it from callers to callees. Finally, if $m = m_{start}$, then each abstract object reachable in the heap abstraction from any abstract object in the points-to set of the 0^{th} actual argument of i is added to the thread-escaping set.

2.5.3 Computation of `escapingRaces`

The result of our thread escape analysis is used to approximate the second of our four conditions for race freedom, namely, that a pair of statements cannot be involved in a race if the memory location accessed by either (or both) of the statements is always thread-local. This computation is specified as Algorithm 2.3.

Algorithm 2.3. Computation of `escapingRaces`.

DOMAINS

$\mathbb{C}$ abstract contexts
 $\mathbb{E}$ statements accessing a field or array element
 $\mathbb{G}$ static fields
 $\mathbb{O}$ abstract objects
 $\mathbb{P}$ program points
 $\mathbb{V}$ local variables

RELATIONS

input $\mathbf{ptsIn} : \mathbb{C} \times \mathbb{P} \times \mathbb{V} \times \mathbb{O}$
 input $\mathbf{escIn} : \mathbb{C} \times \mathbb{P} \times \mathbb{O}$
 input $\mathbf{EG} : \mathbb{E} \times \mathbb{G}$
 input $\mathbf{EV} : \mathbb{E} \times \mathbb{V}$
 input $\mathbf{EP} : \mathbb{E} \times \mathbb{P}$
 output $\mathbf{escapingRaces} : \mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

RULES

$$\mathbf{escapingRaces}(t_1, c_1, e_1, t_2, c_2, e_2) \text{ :- } \mathbf{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \mathbf{EG}(e_1, g), \mathbf{EG}(e_2, g). \quad (2.12)$$

$$\mathbf{escapingRaces}(t_1, c_1, e_1, t_2, c_2, e_2) \text{ :- } \mathbf{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \mathbf{EV}(e_1, v_2), \mathbf{EP}(e_1, p_1), \mathbf{ptsIn}(c_1, p_1, v_1, o_1), \mathbf{escIn}(c_1, p_1, o_1), \mathbf{EV}(e_2, v_2), \mathbf{EP}(e_2, p_2), \mathbf{ptsIn}(c_2, p_2, v_2, o_2), \mathbf{escIn}(c_2, p_2, o_2). \quad (2.13)$$

□

Besides the point-to relation $\mathbf{ptsIn}$ and the thread escape relation $\mathbf{escIn}$, both obtained from our thread escape analysis, the algorithm requires the following basic program relations:

- $\mathbf{EG} : \mathbb{E} \times \mathbb{G}$ contains each tuple (e, g) such that statement e accesses static field g .

- **EV** : $\mathbb{E} \times \mathbb{V}$ contains each tuple (e, v) such that statement e accesses an instance field or array element, and e reads local variable v to refer to the object whose instance field or array element is accessed.
- **EP** : $\mathbb{E} \times \mathbb{P}$ contains each tuple (e, p) such that statement e occurs at program point p .

Rules (2.12) and (2.13) compute **escapingRaces**, a subset of **originalRaces** obtained by eliminating each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \mathbf{originalRaces}$ that is deemed a non-race because either e_1 or e_2 (or both) always access a memory location that is thread-local during the access, in abstract contexts c_1 and c_2 of their containing methods, respectively. (Notice that this computation does not inspect abstract threads t_1 and t_2 in the tuple.) Recall from the computation of **originalRaces** in Algorithm 2.1 that e_1 and e_2 either both access the same instance field or both access array elements or both access static fields; we next consider each of these cases in turn.

Rule (2.12) considers the case in which a static field is accessed. In this case, the tuple cannot be eliminated because static fields in Java are akin to global variables, and our thread escape analysis regards each access to a global variable as thread-shared.

Rule (2.13) considers the case in which an instance field or an array element (that is, the hypothetical field f_{elems}) is accessed. Suppose e_1 and e_2 contain accesses of the form $v_1.f$ and $v_2.f$ at program points p_1 and p_2 , respectively. The rule says that the tuple cannot be eliminated if v_1 and v_2 may point to abstract objects o_1 and o_2 in abstract contexts c_1 and c_2 just before program points p_1 and p_2 , respectively, and both may be thread-shared during the accesses. If this condition does not hold, then the thread escape analysis guarantees that either v_1 or v_2 (or both) never refer to an object that is thread-shared during the access in abstract contexts c_1 and c_2 , respectively. The field f of a thread-local object has a memory location that is thread-local and hence it follows that e_1 and e_2 cannot be involved in a race and the tuple can be eliminated.

2.6 May-Happen-In-Parallel Analysis

In this section, we present our may-happen-in-parallel analysis and its application to our race detection algorithm.

In the literature, the term *may-happen-in-parallel analysis* (also called *MHP analysis*, *concurrency analysis*, and *non-concurrency analysis*) is broadly used for any analysis that approximates the set of statements in a concurrent program that may be executed simultaneously [4, 47, 53, 56, 62, 63]. The analysis typically analyzes the task and synchronization structure of the program. Since different languages have diverse ways for expressing tasks and synchronization, however, the analysis is typically very language-specific.

Our may-happen-in-parallel analysis is tailored to Java, which uses asynchronous threads to express tasks and primarily locks to express synchronization. Furthermore, our analysis is tailored to the application of static race detection, and only models the thread structure of the program, ignoring locks and any other synchronization idioms used, such as fork-join, wait-notify, barrier synchronization, etc. Our race detection algorithm uses a separate analysis for handling lock-based synchronization, namely, the lockset analysis in Section 2.7, which is further improved upon by a so-called conditional must not alias analysis in Chapter 3.

Our may-happen-in-parallel analysis produces relation `parallel` : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C}$ which contains each tuple (t_1, c, e, t_2) such that some thread abstracted by t_1 may execute statement e in abstract context c of e 's containing method while a different thread abstracted by t_2 may be running in parallel.

The relation `parallel` computed for our running example from Figure 2.1 is shown in Figure 2.3. Recall that there are two abstract threads in this example: ϵ abstracting the implicit main thread and `[h3]` abstracting all the explicitly spawned child threads. The first two tuples in relation `parallel` capture the fact that the main thread executes statements `e1` and `e2` in `T.<init>` in the process of creating a new child thread, while the previously spawned child threads abstracted by `[h3]` may be running in parallel. The next four tuples occur because the main thread calls `B.get` and `B.set` on objects abstracted by `[h1]` and `[h2]`, respectively, which call `A.get`

and **A.set** on objects abstracted by $[h4:h1]$ and $[h4:h2]$, respectively. In each of these four cases, child threads abstracted by $[h3]$ may be running parallel. The next six tuples occur because each child thread calls **B.run** on an object abstracted by $[h3]$, which in turn calls **B.get** and **B.set** on objects abstracted by $[h2]$ and $[h1]$, respectively, which call **A.get** and **A.set** on objects abstracted by $[h4:h2]$ and $[h4:h1]$, respectively. In each of these six cases, the main thread may be running in parallel. The last six tuples are similar to the preceding six tuples but capture the fact that another child thread (besides the main thread) may be running in parallel while one child thread executes the denoted statements.

2.6.1 Computation of parallelRaces

The relation **parallel** computed by our may-happen-in-parallel analysis is used to approximate the third of our four conditions for race freedom, namely, that a pair of statements cannot be involved in a race if they are ordered by the thread structure of the program. This computation is specified as Algorithm 2.4. Rule (2.14), the only rule in the algorithm, computes **parallelRaces** as containing each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \text{originalRaces}$ such that a thread abstracted by t_1 may execute e_1 in abstract context c_1 while a thread abstracted by t_2 is running in parallel and, likewise, a thread abstracted by t_2 may execute e_2 in abstract context c_2 while a thread abstracted by t_1 may be running in parallel. If this condition does not hold, our may-happen-in-parallel analysis ensures that any pair of threads abstracted by t_1 and t_2 can never simultaneously execute e_1 and e_2 in abstract contexts c_1 and c_2 , respectively, and hence the tuple is race-free and can be eliminated.

The relation **parallelRaces** computed for our running example from Figure 2.1 is shown in the fourth column of Table 2.1. A $\checkmark$ mark in a particular row in that column indicates that the corresponding tuple in **originalRaces** (shown in the first column of the same row) is retained in **parallelRaces**. For instance, consider tuples #3 and #4. Tuple #4 is retained because **e2** may be executed in abstract context $[h3]$ by the main thread while some child thread is running in parallel and, likewise, **e4** may be executed by some child thread in abstract context $[h3]$ while the main thread is

running in parallel. But tuple #3 is eliminated because **e2** cannot be executed by one thread abstracted by ϵ while a different thread also abstracted by ϵ is running in parallel (note that unlike **[h3]** which abstracts all child threads, ϵ abstracts the lone main thread). In all, this computation alone eliminates 78% (28/36) of the tuples in **originalRaces** in our example.

Algorithm 2.4. Computation of **parallelRaces**.

DOMAINS

$\mathbb{C}$ abstract contexts

$\mathbb{E}$ statements accessing a field or array element

RELATIONS

input **originalRaces** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

input **parallel** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C}$

output **parallelRaces** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

RULES

$$\begin{aligned} \mathbf{parallelRaces}(t_1, c_1, e_1, t_2, c_2, e_2) &:- \mathbf{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\ \mathbf{parallel}(t_1, c_1, e_1, t_2), \mathbf{parallel}(t_2, c_2, e_2, t_1). \end{aligned} \quad (2.14)$$

□

Our may-happen-in-parallel analysis is particularly effective at eliminating two kinds of non-races. First, it eliminates each tuple of the form $(t, c_1, e_1, t, c_2, e_2)$ where t abstracts at most one thread. For instance, in our running example, ϵ abstracts the lone main thread. As a result, the relation **parallel** never contains a tuple of the form $(\epsilon, -, -, \epsilon)$ and so by Rule (2.14) of Algorithm 2.4, the relation **parallelRaces** never contains a tuple of the form $(\epsilon, -, -, \epsilon, -, -)$. Tuple #3 we discussed above illustrates this case.

Secondly, our may-happen-in-parallel analysis is effective at eliminating non-races on static fields. Our running example does not have any static fields but we illustrate

```

parallel :  $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} = \{$ 
    ( $\epsilon$ , [h3], e1, [h3])
    ( $\epsilon$ , [h3], e2, [h3])
    ( $\epsilon$ , [h1], e6, [h3])
    ( $\epsilon$ , [h2], e7, [h3])
    ( $\epsilon$ , [h4::h1], e9, [h3])
    ( $\epsilon$ , [h4::h2], e10, [h3])
    ([h3], [h3], e3,  $\epsilon$ )
    ([h3], [h3], e4,  $\epsilon$ )
    ([h3], [h2], e6,  $\epsilon$ )
    ([h3], [h1], e7,  $\epsilon$ )
    ([h3], [h4::h2], e9,  $\epsilon$ )
    ([h3], [h4::h1], e10,  $\epsilon$ )
    ([h3], [h3], e3, [h3])
    ([h3], [h3], e4, [h3])
    ([h3], [h2], e6, [h3])
    ([h3], [h1], e7, [h3])
    ([h3], [h4::h2], e9, [h3])
    ([h3], [h4::h1], e10, [h3])
 $\}$ 

```

Figure 2.3: May-happen-in-parallel analysis of example program.

this case for another example, shown in Figure 2.4. For this example, there are two abstract threads, namely, ϵ abstracting the lone main thread and [h1] abstracting all child threads, and we have $\text{originalRaces} = \{ (\epsilon, \epsilon, \text{e1}, \epsilon, \epsilon, \text{e1}), (\epsilon, \epsilon, \text{e1}, [\text{h1}], [\text{h1}], \text{e2}) \}$. Neither the may alias analysis nor the thread escape analysis is capable of eliminating either of the two tuples as they are incapable of eliminating races on static fields (recall Rule (2.10) in Algorithm 2.2 and Rule (2.12) in Algorithm 2.3). But our may-happen-in-parallel analysis eliminates both these tuples: the first because it is of the form $(\epsilon, -, -, \epsilon, -, -)$ and ϵ abstracts the lone main thread, and the second because relation `parallel` for this example contains $([\text{h1}], [\text{h1}], \text{e2}, \epsilon)$ but not $(\epsilon, \epsilon, \text{e1}, [\text{h1}])$.

```

public class T {
    private static int g;
    public class main(String[] a) {
        T.g = ...;           // e1 (write to static field T.g)
        while (*) {
            T t = new T();    // h1
            t.start();
        }
    }
    public void run() {
        ... = T.g;           // e2 (read of static field T.g)
    }
}

```

Figure 2.4: Example multithreaded Java program accessing a static field.

2.7 Lockset Analysis

In this section, we present our lockset analysis and its application to our race detection algorithm. This analysis is essentially a static approximation of the *lockset algorithm* [74] used by a class of dynamic race detectors. The lockset algorithm is tailored to the common lock-based synchronization discipline: a memory location is race-free if every thread holds some common lock while accessing that location.

Our lockset analysis computes relation **guarded** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C}$ containing each tuple (t, c, e, o) such that each thread abstracted by t may hold a lock on some object abstracted by o while executing statement e in abstract context c of e 's containing method.

The relation **guarded** computed for our running example from Figure 2.1 is shown in Figure 2.5. The lockset analysis computes this relation by simulating the execution of each abstract thread on the context sensitive call graph of the program constructed by our k -object-sensitive analysis, accumulating the set of abstract locks held by each abstract thread while executing each statement. Recall that there are two abstract threads in our example, namely, ϵ abstracting the main thread and $[h3]$ abstracting all child threads; we consider each of these in turn.

```

guarded :  $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} = \{$ 
    ( $\epsilon$ , [h2], e7, [h2])
    ( $\epsilon$ , [h4::h2], e10, [h2])
    ([h3], [h2], e6, [h2])
    ([h3], [h4::h2], e9, [h2])
 $\}$ 

```

Figure 2.5: Lockset analysis of example program.

We first consider abstract thread ϵ . The analysis begins simulating the execution of method `T.main` in abstract context ϵ as this is the root method of the main thread. Note that the main thread acquires a lock on the object pointed to by local variable `v2` when it executes the statement labeled 11. The analysis uses points-to information computed by our k -object-sensitive analysis to infer that `v2` may point to abstract lock [h2] in abstract context ϵ (recall that relation `ptsV` for this example, shown in Figure 2.2, contains the tuple $(\epsilon, \text{v2}, [\text{h2}])$). The analysis then follows call site [i6] and descends into method `B.set` in abstract context [h2] while holding abstract lock [h2], where it encounters statement `e7` and adds tuple $(\epsilon, [\text{h2}], \text{e7}, [\text{h2}])$ to relation `guarded`. It next follows call site i11 and descends into method `A.set` in abstract context [h4::h2] while still holding abstract lock [h2], where it encounters statement `e10` and adds tuple $(\epsilon, [\text{h4}::\text{h2}], \text{e10}, [\text{h2}])$ to relation `guarded`.

Next consider abstract thread [h3]. The analysis begins simulating the execution of method `T.run` in abstract context [h3] as this is the root method of each child thread. Each child thread acquires a lock on the object pointed to by local variable `v7` when it executes the statement labeled 12. The analysis uses points-to information computed by our k -object-sensitive analysis to infer that `v7` may point to abstract lock [h2] in abstract context [h3] (recall that relation `ptsV` for this example, shown in Figure 2.2, contains the tuple $([\text{h3}], \text{v7}, [\text{h2}])$). Similar to the case of the main thread, the analysis then proceeds to add tuples $([\text{h3}], [\text{h2}], \text{e6}, [\text{h2}])$ and $([\text{h3}], [\text{h4}::\text{h2}], \text{e9}, [\text{h2}])$ to relation `guarded`.

2.7.1 Computation of `unlockedRaces`

The relation `guarded` is used to approximate the final of our four conditions for race freedom, namely, that a pair of statements cannot be involved in a race if they are ordered by lock-based synchronization. This computation is specified as Algorithm 2.5. Rules (2.15) and (2.16) compute intermediate relation `unlikelyRaces` as containing each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \text{originalRaces}$ that satisfies any of the following two conditions:

1. the pair of threads abstracted by t_1 and t_2 *may* hold a common lock abstracted by o while executing statements `e1` and `e2`, respectively.
2. the pair of threads abstracted by t_1 and t_2 *may* be one and the same.

Then, Rule (2.17) computes relation `unlockedRaces` as containing each tuple in `originalRaces` that satisfies neither of the above conditions. This computation of `unlockedRaces`, however, is unsound: if a tuple is not retained in `unlockedRaces`, then it is not necessarily the case that the tuple is race-free, because condition (1) merely requires that the pair of threads *may* hold a common lock, whereas for soundness, it should insist that the threads *must* hold a common lock and, similarly, condition (2) merely requires that the pair of threads *may* be the same, whereas for soundness, it should insist that the threads *must* be the same. In Chapter 3, we replace this unsound lockset analysis with *conditional must not alias analysis* which computes conditions (1) and (2) soundly.

The relation `unlockedRaces` computed for our running example from Figure 2.1 is shown in the fifth column of Table 2.1. A $\checkmark$ mark in a particular row in that column indicates that the corresponding tuple in `originalRaces` (shown in the first column of the same row) is retained in `unlockedRaces`. For instance, consider tuples #32, #34, and #35. Tuple #34 is retained because it satisfies neither condition (1) (clearly no common lock is held when the main thread executes `e9` in abstract context `[h4::h1]` and a child thread executes `e10` in abstract context `[h4::h1]`) nor condition (2) (the main thread is clearly distinct from any child thread). But tuple #35 is eliminated because it satisfies condition (1) above, namely, a common

lock may be held when a child thread executes **e9** in abstract context $[h4:h2]$ and the main thread executes **e10** in abstract context $[h4:h2]$. Likewise, tuple #32 is eliminated because it satisfies condition (2) above, namely, the pair of child threads abstracted by $[h3]$ may be one and the same. In all, this computation alone eliminates 64% (23/36) of the tuples in **originalRaces** in our example. As we explained above, however, this computation is unsound in that the eliminated tuples are not necessarily race-free. For instance, of the tuples discussed above, the elimination of tuple #35 is correct because a common lock is indeed held, but the elimination of tuple #32 is incorrect because two different child threads can simultaneously write to the same shared integer. In fact, tuple #32 is a real race that our algorithm fails to report.

Algorithm 2.5. Computation of **unlockedRaces**.

DOMAINS

$\mathbb{C}$ abstract contexts

$\mathbb{E}$ statements accessing a field or array element

RELATIONS

input **originalRaces** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

input **guarded** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C}$

output **unlockedRaces** : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

RULES

$$\begin{aligned} \text{unlikelyRaces}(t_1, c_1, e_1, t_2, c_2, e_2) &:- \text{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\ &\quad \text{guarded}(t_1, c_1, e_1, o), \text{ guarded}(t_2, c_2, e_2, o). \end{aligned} \quad (2.15)$$

$$\begin{aligned} \text{unlikelyRaces}(t_1, c_1, e_1, t_2, c_2, e_2) &:- \text{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\ &\quad t_1 = t_2. \end{aligned} \quad (2.16)$$

$$\begin{aligned} \text{unlockedRaces}(t_1, c_1, e_1, t_2, c_2, e_2) &:- \text{originalRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\ &\quad \neg \text{unlikelyRaces}(t_1, c_1, e_1, t_2, c_2, e_2). \end{aligned} \quad (2.17)$$

□

2.8 Putting it all together

We have presented four static analyses, namely, a may alias analysis, a thread escape analysis, a may-happen-in-parallel analysis, and a lockset analysis, and shown how each of these analyses approximates a different condition for race freedom to prune `originalRaces`, our initial set of potential races in the given Java program. In this section, we show how to put these analyses together to compute `ultimateRaces`, the final set of potential races that is output by our race detection algorithm. The computation is presented in Algorithm 2.6.

Algorithm 2.6. Computation of `ultimateRaces`.

DOMAINS

- $\mathbb{C}$ abstract contexts
- $\mathbb{E}$ statements accessing a field or array element
- $\mathbb{F}$ instance fields
- $\mathbb{O}$ abstract objects
- $\mathbb{P}$ program points
- $\mathbb{V}$ local variables

RELATIONS

- input `parallelRaces` : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$
- input `unlockedRaces` : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$
- input `ptsIn` : $\mathbb{C} \times \mathbb{P} \times \mathbb{V} \times \mathbb{O}$
- input `escIn` : $\mathbb{C} \times \mathbb{P} \times \mathbb{O}$
- input `EG` : $\mathbb{E} \times \mathbb{G}$
- input `EV` : $\mathbb{E} \times \mathbb{V}$
- input `EP` : $\mathbb{E} \times \mathbb{P}$
- output `ultimateRacesWithObject` : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{O}$
- output `ultimateRaces` : $\mathbb{C} \times \mathbb{C} \times \mathbb{E} \times \mathbb{C} \times \mathbb{C} \times \mathbb{E}$

RULES

$$\begin{aligned}
&\text{ultimateRacesWithObject}(t_1, c_1, e_1, t_2, c_2, e_2, \epsilon) :- \\
&\quad \text{parallelRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \text{ unlockedRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\
&\quad \text{EG}(e_1, g), \text{ EG}(e_2, g). \tag{2.18}
\end{aligned}$$

$$\begin{aligned}
&\text{ultimateRacesWithObject}(t_1, c_1, e_1, t_2, c_2, e_2, o) :- \\
&\quad \text{parallelRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \text{ unlockedRaces}(t_1, c_1, e_1, t_2, c_2, e_2), \\
&\quad \text{EV}(e_1, v_1), \text{ EP}(e_1, p_1), \text{ ptsIn}(c_1, p_1, v_1, o), \text{ escIn}(c_1, p_1, o), \\
&\quad \text{EV}(e_2, v_2), \text{ EP}(e_2, p_2), \text{ ptsIn}(c_2, p_2, v_2, o), \text{ escIn}(c_2, p_2, o). \tag{2.19}
\end{aligned}$$

$$\begin{aligned}
&\text{ultimateRaces}(t_1, c_1, e_1, t_2, c_2, e_2) :- \\
&\quad \text{ultimateRacesWithObject}(t_1, c_1, e_1, t_2, c_2, e_2, -). \tag{2.20}
\end{aligned}$$

□

Rules (2.18) and (2.19) compute relation `ultimateRacesWithObject` as containing each tuple $(t_1, c_1, e_1, t_2, c_2, e_2, o)$ such that there is a potential race on a field of abstract object o accessed by statements e_1 and e_2 in abstract contexts c_1 and c_2 by abstract threads t_1 and t_2 , respectively. We compute this relation because we need the sets of abstract objects on whose fields potential races occur, for two reasons. First, we wish to group race reports by these sets of abstract objects (we explain in Section 2.9.1 why such a grouping is useful). Secondly, recall that in this chapter, we have presumed a non-adaptive k -object-sensitive analysis that uses the same k value for all object allocation sites in the given Java program, which affects precision if a small k value is used, and affects scalability if a large k value is used. In Chapter 4, we present an adaptive k -object-sensitive analysis capable of using different k values for different object allocation sites in the same program. It enables a demand-driven race detection algorithm that uses bigger k values for a few sites and smaller k values for the vast majority of sites and thereby strikes a good trade-off between scalability and precision. The demand-driven algorithm is iterative, and runs the current race detection algorithm in each iteration but it uses a k -object-sensitive analysis with $k = 1$ for all object allocation sites in the first iteration, and it uses the sets

of abstract objects on whose fields potential races are reported in iteration $N - 1$ to determine what k value to use for each object allocation site in iteration N .

Returning to our computation of relation `ultimateRacesWithObject`, recall that for any tuple we consider, e_1 and e_2 either both access the same instance field or both access array elements or both access the same static field; we consider each of these cases in turn.

Rule (2.18) considers the case in which a static field is accessed. It states that `ultimateRacesWithObject` contains each tuple that could not be eliminated by both the may-happen-in-parallel analysis and the lockset analysis (recall that neither our may alias analysis nor our thread escape analysis can eliminate potential races on static fields). We regard ϵ as the hypothetical abstract object on whose field the race occurs.

Rule (2.19) considers the case in which an instance field or an array element (that is, the hypothetical field f_{elems}) is accessed. It states that `ultimateRacesWithObject` contains each tuple that could not be eliminated by any of our four static analyses. It explicitly uses the results of the may-happen-in-parallel analysis and the lockset analysis (that is, relations `parallelRaces` and `unlockedRaces`, respectively) but it uses the results of the may alias analysis and the thread escape analysis implicitly, as follows. Suppose accesses e_1 and e_2 are of the form $v_1.f$ and $v_2.f$, respectively, where v_1 and v_2 are local variables and f is an instance field or the hypothetical field f_{elems} . Let O_1 and O_2 be the thread-escaping subsets of the points-to sets of v_1 and v_2 , respectively. Then, the rule states that if there is indeed a race between e_1 and e_2 , then the race must occur on field f of an object abstracted by some object in $O_1 \cap O_2$.

Finally, rule (2.20) computes relation `ultimateRaces` as containing each $(t_1, c_1, e_1, t_2, c_2, e_2)$ such that there is some abstract object o such that $(t_1, c_1, e_1, t_2, c_2, e_2, o) \in \text{ultimateRacesWithObject}$.

Our race detection algorithm is not sound primarily because it uses an unsound lockset analysis: if a tuple is eliminated from `ultimateRaces` only because of the lockset analysis (that is, the tuple does not occur in `unlockedRaces` but occurs in each of `aliasingRaces`, `escapingRaces`, and `parallelRaces`), then the tuple may

not be race-free (that is, it may be a *false negative*). Otherwise, the eliminated tuple is guaranteed to be race-free. Our race detection algorithm is also not complete in that a tuple retained in `ultimateRaces` may not be a real race (that is, it may be a *false positive*).

The relation `ultimateRaces` for our running example from Figure 2.1 is shown in the last column of Table 2.1. A ✓ mark in a particular row in that column indicates that the corresponding tuple in `originalRaces` (shown in the first column of the same row) is retained in `ultimateRaces`. This example does not have any false positives: tuple #34, the only tuple retained, is indeed a real race because the main thread executes `e9`, which reads from a heap location, and each child thread executes `e10`, which writes to the same location, and there is no ordering enforced between the two accesses. However, this example has a false negative: tuple #32 is eliminated only due to the lockset analysis, which infers that two different child threads may not execute `e10` to write to the same heap location, and therefore the tuple is likely to be race-free. The tuple is not race-free, however, because different child threads do in fact write to the same location without any ordering enforced between the writes. Finally, note that tuple #35 is also eliminated only due to the lockset analysis, which infers that a common lock may be held when the main thread executes `e10`, which writes to a heap location, and a child thread executes `e9`, which reads from the same location, and therefore the tuple is likely to be race-free. The tuple is indeed race-free, because a common lock is in fact held when the main thread writes to the location and each child thread reads from that location.

2.9 Usability Issues

In this section, we address usability issues in our race detection algorithm, namely, how it generates counterexamples to explain the detected races and how it checks open programs such as libraries.

2.9.1 Counterexamples

Our algorithm outputs each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \text{ultimateRaces}$ as a potential race. Reporting races found by a static race detection tool in a useful manner poses several challenges:

- Since a race involves a pair of accesses, there is a potentially quadratic blowup in the output of the tool.
- Races are symptoms as opposed to causes of bugs. Thus, a single race may indicate multiple bugs and, conversely, multiple races may indicate the same bug.
- Determining whether a reported race manifests a real race or a false positive involves manually inspecting various aspects such as the memory location on which the race occurs, the pair of threads executing the statements involved in the race, and the call stack and the set of locks held by either thread at the point of the race.
- Even if a reported race is real, manual inspection is needed to determine whether it is a harmful race, that is, a violation of a program invariant, or a benign race. The problem is exacerbated by the subtleties of the Java memory model [55].

We address the above issues by grouping related race reports together and reporting counterexamples along with each race report.

Our algorithm reports each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \text{ultimateRaces}$ in one of three categories, namely, *races on instance fields*, *races on array elements*, and *races on static fields*, depending upon whether e_1 and e_2 access an instance field, array elements (that is, the hypothetical field f_{elems}), or a static field, respectively. Within each of these categories, we provide two views: a *field-based view* that groups race reports in the category by the field on which the race occurs, and a *object-based view* that groups race reports in the category by the set of abstract objects on whose field the race occurs. However, the field-based view is not useful for the category of races on array elements as all races in this category are on the same hypothetical field f_{elems} .

and therefore would belong to a single group in the field-based view. Likewise, the object-based view is not useful for the category of races on static fields as all races in this category are on the singleton set consisting of the hypothetical abstract object ϵ and therefore would belong to a single group in the object-based view.

The field-based view is the conventional grouping approach used by previous race detection techniques. It helps in, for instance, quickly ignoring all intentional races on a field that are benign because the field is declared volatile (e.g., an integer-valued field that tracks statistics approximately, a boolean field that one thread periodically polls and another writes to notify it, etc.) Our object-based view is novel, and it is useful for:

- quickly identifying all false positives arising from a single source of imprecision in our may alias analysis or thread escape analysis (such races are scattered over multiple groups in the field-based view but belong to the same group in the object-based view)
- reporting races on elements of different arrays in separate groups (as we mentioned above, all such races belong to a single group in the field-based view)
- reporting races triggered on the same instance field by different clients in separate groups, for instance, if a class C has an instance field f and there are two different sites h_1 and h_2 allocating objects of class C , then reporting races triggered on field $C.f$ of objects allocated at sites h_1 and h_2 in separate groups.

Besides grouping related races together, our algorithm reports various aspects for each $(t_1, c_1, e_1, t_2, c_2, e_2) \in \text{ultimateRaces}$, namely, the field on which the race occurs, the set of abstract objects on whose field the race occurs, the pair of statements e_1 and e_2 involved in the race, and the abstract threads t_1 and t_2 that execute those statements. It also reports each pair of cycle-free paths in the context sensitive call graph of the program, obtained from the k -object-sensitive analysis, starting from the root methods of abstract threads t_1 and t_2 in abstract contexts t_1 and t_2 , respectively, and terminating in the containing methods of e_1 and e_2 in abstract contexts c_1 and c_2 , respectively. Each path denotes a possible call stack of the corresponding

thread at the point of the race. Along with each path is also specified the set of abstract locks held by the thread along that path at the point of the race.

2.9.2 Open Programs

Our race detection algorithm performs a whole-program analysis and therefore needs a complete Java program. Many multithreaded Java programs, however, are open programs (e.g., libraries) that provide an interface to interact with their environment. It is beneficial to detect races in such programs before deploying them in specific environments. There are two problems with analyzing open programs: missing callee code (e.g., native methods in Java) and missing caller code (e.g., a missing main method in the case of a library).

We model missing callees using “stubs” in place of commonly used native methods in the Java standard library (e.g., the `start` method of class `java.lang.Thread`) and treat all other missing and native methods unsoundly as no-ops. We model missing callers by automatically synthesizing a harness that simulates many scenarios in which the environment might exercise the program’s interface. Due to the complexity of the Java language, our current harness synthesis algorithm is not sound in that it does not simulate all possible scenarios. Nevertheless, it is much easier in principle to construct a sound harness for our technique than for a model checker that also performs whole-program analysis, because model checkers are typically path sensitive whereas none of the static analyses used in our technique is path sensitive and so the harness required by our technique need not be as elaborate.

Our harness synthesis algorithm takes as input an open Java program and a user-specified set of public interfaces or classes in the program, denoted $\mathcal{I}$, and builds a class `Harness` that:

1. declares a private static field of each type allowed as an argument type or return type of any public method declared in any interface or class in $\mathcal{I}$, and
2. declares a public static main method that:

- (a) non-deterministically assigns to each instance field of reference type generated in step (1) above an object of any public concrete class in the program having a compatible type, by invoking any public constructor method(s) of that class, and
- (b) executes a loop whose body non-deterministically invokes in a fresh child thread any public non-constructor method declared in any interface or class in $\mathcal{I}$ on any combination of instance fields generated in step (1) that respects the argument types of the method and assigns the return value to any instance field generated in step (1) that respects the return type of the method.

For our running example from Figure 2.1, suppose class `T` is missing and only classes `A` and `B` are provided, and suppose $\mathcal{I} = \{\text{B}\}$. Then, our harness synthesis algorithm generates the harness shown in Figure 2.6. It is easy to see that our race detection algorithm reports the following two races in the resulting program, both on field `f4` of class `A`:

- A race between two different threads allocated at site `ht1`. Both threads call method `B.set` which in turn calls method `A.set` writes to field `f4`. Both threads operate on the same `B` object allocated at site `hb` and hence write to the same memory location denoted by field `f4`.
- A race between a thread allocated at site `ht1` and a thread allocated at site `ht2`. The former thread calls method `B.set` which in turn calls method `A.set` which writes to field `f4` while the latter thread calls method `B.get` which in turn calls method `A.get` which reads field `f4`. Again, both threads operate on the same `B` object allocated at site `hb` and hence access the same memory location denoted by field `f4`.

Both races are real and indicate that class `B` is not thread-safe, that is, clients of the class must use their own synchronization whenever they call method `B.set` concurrently on the same `B` object from different threads and whenever they call methods `B.set` and `B.get` concurrently on the same `B` object from different threads.


```
import java.lang.Thread;

public class Harness {
    private static int i;
    private static B b;
    public static void main(String[] args) {
        if (*) {
            b = new B();
        }
        while (*) {
            if (*) {
h1:                Thread t = new Thread {
                    public void run() { i = b.get(); }
                    };
                    t.start();
            }
            if (*) {
h2:                Thread t = new Thread {
                    public void run() { b.set(i); }
                    };
                    t.start();
            }
        }
    }
}
```

Figure 2.6: Example harness.

	# classes	# bytecodes	brief description
<code>elevator</code>	1066	109831	Discrete event simulator
<code>tsp</code>	1068	110582	TSP solver from ETH
<code>hedc</code>	1592	278010	Web crawler from ETH
<code>ftp</code>	1905	348813	Apache FTP server
<code>pool</code>	1121	122187	Apache pooling library
<code>jdbf</code>	1739	291392	Object-relational mapping system
<code>jtds</code>	1801	301231	JDBC driver
<code>derby</code>	3428	721912	Apache relational database engine

Table 2.2: Benchmarks.

2.10 Unsoundness

Our race detection algorithm has the following three sources of unsoundness:

1. It uses an unsound lockset analysis to check whether a pair of accesses is ordered by lock-based synchronization (see Section 2.7). In Chapter 3, we replace this unsound lockset analysis with a sound *conditional must not alias analysis*.
2. It does not analyze open programs soundly (see Section 2.9.2). We hope to address sound race detection for open programs in future work.
3. It ignores the effects of dynamic class loading and reflection.

2.11 Experiments

We have implemented our race detection algorithm in a tool **Chord** and applied it to a suite of eight multithreaded Java programs. Table 2.2 provides a brief description of each program along with the number of classes and the number of bytecodes in the program. The numbers correspond to code that is deemed reachable from the main method of each program in a context insensitive call graph that is computed by Spark [52], a 0-CFA-based may alias analysis with on-the-fly call graph construction provided in the Soot compiler framework [81]. The experiments were performed on a 2.4GHz machine with 4GB memory.

benchmark	running time
<code>elevator</code>	7m11s
<code>tsp</code>	6m04s
<code>hedc</code>	15m13s
<code>ftp</code>	22m24s
<code>pool</code>	8m02s
<code>jdbf</code>	11m45s
<code>jtds</code>	15m39s
<code>derby</code>	46m29s

Table 2.3: Experimental results: Running time.

The benchmark suite includes four closed programs: `elevator`, `tsp`, `hedc`, and `ftp`. The remaining four benchmarks `pool`, `jdbf`, `jtds`, and `derby` are open programs that provide interfaces to interact with their environments. Benchmark `elevator` is a discrete event simulator program, `tsp` is an implementation of the Traveling Salesman Problem from ETH Zurich, `hedc` is a web crawler from ETH Zurich, `ftp` is the Apache FTP Server, `pool` is Apache Commons Pool (a generic object-pooling library), `jdbf` denotes JdbF (an object-relational mapping system which simplifies the work of retrieving, saving, and deleting objects from a relational database), `jtds` denotes jTDS (the fastest open-source JDBC driver for Microsoft SQL Server and Sybase), and `derby` is Apache Derby (a relational database management system). Benchmarks `elevator`, `tsp`, and `hedc` have been analyzed in previous work on race detection; the rest are mature and widely used programs, except for `jdbf`, which is in its developmental stages.

Table 2.3 presents the total running time of `Chord` for each benchmark. Recall from Section 2.4 that our implementation of k -object-sensitive analysis is parameterized by a positive integer k that may be instantiated differently for different programs; we have used $k = 3$ for each experiment.

Table 2.4 presents the sizes of the initial and final sets of potential races as well as the contribution of each of the four static analyses. The “original” and “ultimate” columns presents the sizes of sets $\pi_{3,6}(\text{originalRaces})$ and $\pi_{3,6}(\text{ultimateRaces})$, respectively, that is, the number of original and ultimate race pairs. These sets are

benchmark	original	% of original pairs deemed race-free				ultimate
		aliasing	escaping	parallel	unlocked	
elevator	139850	76%	74%	52%	39%	0
tsp	155094	75%	71%	51%	41%	8
hedc	420656	72%	73%	48%	43%	143
ftp	707898	67%	69%	43%	48%	247
pool	205625	73%	70%	46%	45%	27
jdbf	516906	69%	62%	45%	40%	266
jtds	651923	65%	53%	41%	47%	248
derby	1261750	72%	64%	39%	42%	1018

Table 2.4: Experimental results: Numbers of race pairs.

benchmark	ultimate racing pairs			bugs
	real		false	
	harmful	benign		
elevator	0	0	0	0
tsp	6	0	2	0
hedc	107	17	19	4
ftp	199	14	34	32
pool	24	3	0	17
jdbf	258	0	8	18
jtds	227	14	7	16
derby	1018	0	0	319

Table 2.5: Experimental results: Classification of reported races.

obtained by projecting away abstract threads c_1^t and c_2^t and abstract contexts c_1^m and c_2^m from each tuple $(c_1^t, c_1^m, e_1, c_2^t, c_2^m, e_2)$ in sets **originalRaces** and **ultimateRaces**, respectively. The “aliasing”, “escaping”, “parallel”, and “unlocked” columns denote the fraction of original race pairs that were deemed race-free by our may alias analysis, thread escape analysis, may-happen-in-parallel analysis, and lockset analysis, respectively. The may alias analysis and thread escape analysis are most effective, eliminating around 60-70% of original race pairs, while the may-happen-in-parallel analysis and lockset analysis eliminate around 40-50% of original race pairs. Note, however, that the four analyses eliminate different subsets of original race pairs, since the number of ultimate race pairs for each benchmark is less than 0.001% of the number of original race pairs.

Table 2.5 classifies each ultimate race pair as a real race or a false positive, and further classifies each real race as harmful or benign. The classification is done by manually inspecting the counterexamples generated by **Chord**. Finally, the “bugs” column reports the number of distinct fixes that were needed in the source code to eliminate all harmful races. Most fixes involved one of the following: (1) adding synchronization to a piece of code where none existed, (2) extending the scope of an existing synchronized block, (3) changing the expression on which the lock was held by a synchronized block, (4) declaring a field volatile, or (5) removing synchronization because one of the above rendered it redundant (letting it remain could degrade performance or introduce deadlocks). In many cases, a harmful race was triggered in code far apart from the code where synchronization was needed to eliminate the race, for instance, we found many harmful races in library code that were eliminated by adding synchronization to application code. The false positives are primarily due to the fact that our race detection algorithm does not track the values of primitives such as integers and booleans. We next describe each experiment briefly.

Of the programs that have been analyzed in previous work on race detection, **Chord** did not report any harmful races in **elevator** and **tsp**, but it reported 107 harmful races in **hedc**. The harmful races in **hedc** indicate four bugs of which only one has been reported in previous work. The remaining three bugs are due to races triggered in library code by application code, in particular, the application code stores

in each of three static fields a different object of library class `java.util.Calendar` and spawns multiple threads which simultaneously invoke seemingly read-only library methods on the three objects, but these methods in reality mutate the state of the objects and result in races if application methods calling them do not use appropriate synchronization. The reason previous race detectors missed these bugs is that most of them do not analyze library code, typically for scalability reasons, e.g., static race detectors routinely elide checking library code while dynamic race detectors routinely elide instrumenting library code.

Chord reported 199 harmful races in Apache FTP Server (**ftp**) that revealed 32 distinct bugs, of which 11 bugs were fixed within a day of reporting. Another bug that was reported was acknowledged by developers but was not fixed because doing so involves making widespread changes. The remaining 20 bugs were not reported at the time because they eluded detection by an early, buggy version of **Chord**.

Chord reported 24 harmful races in Apache Commons Pool (**pool**), a generic object-pooling library that provides five reference implementations. The races exposed bugs in each of the five implementations, for a total of 17 distinct bugs. All bugs were fixed and five patches were released in less than a week from reporting the bugs. The developers deemed the bug fixes significant enough to release a new version of the library.

Chord reported 227 harmful races revealing 16 distinct bugs in the **jTDS** JDBC driver (**jtds**). The developers initially expressed concerns about degrading performance and introducing deadlocks in the process of fixing the bugs in what they said was fairly mature code that seems to work well enough for most people. However, the seriousness of the bugs manifested in the counterexamples reported by our tool led them to conclude that it was dangerous to let the races lurk, and they fixed all of them and released a patch.

Finally, in our single largest benchmark Apache Derby (**derby**), **Chord** reported 1018 races revealing 319 distinct bugs. The developers acknowledged the bugs, requested us to file bug reports, and promised to look at them in detail in the future. They also inquired about the possibility of running our tool regularly on their source code to prevent new races from being introduced over time.

2.12 Related Work

In this section, we discuss related work, including dynamic race detection techniques (Section 2.12.1), static race detection techniques (Section 2.12.2), and recent work on atomicity (Section 2.12.3).

2.12.1 Dynamic Race Detection

Race detection tools are predominantly dynamic. State-of-the-art dynamic race detectors are precise and scalable. Like any dynamic analysis, however, they are inherently unsound and explore only a fraction of the space of the program’s inputs and thread schedules. Furthermore, they cannot be applied to open programs such as libraries, since such programs cannot be executed in the absence of client code.

Dynamic race detectors may be broadly classified into *happens-before-based*, *lockset-based*, and *hybrid*. Happens-before-based dynamic race detectors [1, 19, 20, 22, 59, 71, 75] are based on Lamport’s happens-before relation [49] which is a partial order on all events of all threads in a concurrent execution such that if a pair of accesses performed by a pair of threads on a memory location are not ordered by this relation, then they are deemed to be involved in a race because there exists a concurrent execution in which they can occur simultaneously. The key problems with happens-before-based race detection are that it is difficult to implement efficiently and, although it produces no false positives, it produces many false negatives.

The lockset algorithm is tailored to the common lock-based synchronization discipline: a race is deemed to occur on a memory location if a common lock is not held by each thread while accessing the location. The original implementation in the Eraser tool [74] incurred a slow-down of 10–30X but several static and dynamic optimization techniques, e.g., [2, 64, 83, 84], have reduced it significantly, with a recent implementation having a run-time overhead of only 13–42% [17]. The primary problems with lockset-based race detection are that it produces many false positives when synchronization idioms other than lock-based synchronization are used, as well as having the usual potential for false negatives of any dynamic analysis.

Dinning and Schonberg [23] first proposed combining happens-before-based and

lockset-based race detection (in fact, they originated the lockset-based approach in order to improve the happens-before-based approach). Since then, several hybrid techniques have been proposed that gain the benefits of both approaches without suffering the disadvantages of either [42, 65, 68, 92].

2.12.2 Static Race Detection

Static race detectors are either primarily flow insensitive type-based systems [11, 12, 28, 29, 69, 73], flow sensitive static versions of the lockset algorithm [18, 26, 79], or path sensitive model checkers [45, 70].

The most closely related work is that of Choi et al. [18]. Their approach has the same basic inspiration as ours: using a combination of static analyses, the pairs of statements potentially involved in a race can be filtered to a precise set. Also, at a high level, the static analyses they use are similar to ours (may alias analysis, thread escape analysis, etc.). However, their implementation apparently was never applied beyond small Java programs, most likely because their algorithm is context and object insensitive, whereas we have found context and object sensitivity central to producing precise results; indeed, they conclude that even for small Java programs, more precise analysis is needed. Also, they do not address the other aspects of usability, such as handling open programs and reporting counterexamples.

Two static versions of the lockset algorithm we are aware of for C are Warlock [79] and RacerX [26]. Warlock does not trace paths through loops or recursive functions while RacerX targets operating systems code and uses unsound heuristics specific to such code to determine which locks guard which accesses, which code is multithreaded, and which unguarded accesses are benign.

Type-based and model-checking-based approaches to race detection are appealing in part because of their ability to check open programs and to produce counterexamples, respectively. Our approach possesses both these abilities. The key difference between our approach and the type-based ones is that the latter focus on specifying the synchronization discipline by means of types. Inferring this information automatically has proven difficult and, although significant advances have been made, it

remains an active area of research [3, 30, 32, 73]. The key difference between our approach and the model-checking-based ones is that the latter are typically path sensitive. As a result, they can handle various synchronization idioms whereas we handle only lock-based synchronization and fork-based synchronization. However, path sensitivity, besides affecting scalability, tends to limit model checkers to closed programs since they require an elaborate harness for open programs.

Finally, our technique finds more bugs than all previous static race detection techniques. RacerX [26] found 16 bugs in all in two operating systems (Linux comprising 1.8 MLOC and “System X” comprising 500 KLOC) which, to the best of our knowledge, is the largest number of bugs reported in any previous work on static race detection. Tools like RacerX handle large programs but apparently imprecisely so that not many bugs are found. The other class of static tools, including those based on type systems and model checking, perform a relatively precise analysis but have only been applied to small programs. However, there are just not many bugs to find in such programs, as the results of these tools, many of which are sound, indicate.

Unlike the above techniques which address lock-based synchronization and fork-based synchronization, Aiken and Gay [5] address barrier-based synchronization, namely, they present an effect inference system for statically checking whether SPMD programs are free of barrier synchronization problems.

Other static approaches to preventing races include language-based ones such as nesC for C and Guava for Java, which we describe next.

NesC [37] is a race-free subset of C for networked embedded systems. It classifies code as either *asynchronous code* (AC), meaning reachable from at least one interrupt handler, or *synchronous code* (SC), meaning reachable only from *tasks*. Tasks (but not interrupt handlers) are guaranteed to execute without interruption. Hence, unguarded accesses to shared memory are allowed in SC. However, in AC, such accesses must occur only in *atomic sections*, which are guaranteed to execute without interruption.

Guava [7] is a race-free dialect of Java. It distinguishes three kinds of data: *monitors*, which are thread-shared and are always accessed by synchronized methods; *values*, which cannot have references and hence are thread-local; and *objects*, which

can have multiple references but only from within a single thread and hence are thread-local as well. Guava provides type annotations and type rules for stating and statically checking the above kinds of data.

2.12.3 Atomicity Checking

Recent work on verification of shared-memory multithreaded programs has focused on checking *atomicity* [2, 27, 31, 34–36, 73, 85, 86]. Atomicity introduces a specification of the form “**atomic** { *s* }” which states that code fragment *s* is *atomic* if every run of the program in which the execution of *s* by any thread *t* is interrupted by the actions of other threads is equivalent to a run in which the execution of *s* by thread *t* is uninterrupted. Atomicity checkers then check whether each such specification in the given program is indeed atomic.

A more recent idea, called *atomic sets* [82], introduces a specification of the form “**atomic** (*D*) { *s* }” which states that code fragment *s* is atomic with respect to the chunk of data denoted by *D* (called an *atomic set*). Unlike the atomicity construct which is code-centric in that it only specifies a code fragment *s*, the atomic sets construct is data-centric in that it also specifies the chunk of data *D* with respect to which *s* is atomic. The atomicity specification can be viewed as a special case of the atomic sets specification, namely, “**atomic** { *s* }” can be viewed as specifying that *s* is atomic with respect to the entire program data. Thus, atomic sets are more flexible, allowing different code fragments to be atomic with respect to different chunks of data.

The motivation behind atomicity and atomic sets is that race freedom is neither sound nor complete: the presence of races does not necessarily indicate the presence of concurrency bugs (so-called *benign races*) and the absence of races does not necessarily indicate the absence of concurrency bugs. However, we believe race detection is important because of the following reasons:

1. Race freedom is a natural property for existing languages. Atomicity requires programmers to specify which code fragments are intended to be atomic, and atomic sets additionally require programmers to specify the chunks of data with

respect to which the code fragments are intended to be atomic. In the absence of such specifications, atomicity checkers may check whether every synchronized block “`synchronized (l) { s }`” in the program is atomic (in the hope that programmers use lock `l` to make `s` atomic), while atomic sets checkers may check whether instance methods of a class are atomic with respect to instance fields of the class (in the hope that programmers intend methods of a class to atomically manipulate its encapsulated data). However, such automatically inferred specifications have the same shortcoming as race freedom: their violations do not necessarily indicate concurrency bugs and, conversely, the lack of their violations does not necessarily indicate the absence of concurrency bugs.

2. Many concurrency bugs manifested as violations of atomicity and atomic sets specifications are also manifested as races. As a result, from the perspective of finding bugs in existing programs that lack these specifications, checking for races is a viable alternative.
3. Race detection is a first step in many atomicity checkers. In particular, atomicity checkers based on Lipton’s theory of reduction [54] must show that each statement accessing a memory location is both a *left mover* and a *right mover*, which is done by proving the absence of races on that location. The notion of a race is so fundamental that race detection techniques are likely to be leveraged also in checkers for the more recent and more natural atomic sets specification.

Chapter 3

Conditional Must Not Aliasing

Race detection algorithms for shared-memory multithreaded programs using lock-based synchronization must correlate locks with the memory locations they guard. The heart of a proof of race freedom for such programs is showing that if two locks are distinct, then the memory locations they guard are also distinct. This is an example of a general property we call *conditional must not aliasing*: under the assumption that two objects are not aliased, prove that two other objects are not aliased. This chapter introduces the conditional must not aliasing property, presents an analysis that conservatively approximates the property in the context of static race detection for Java, and demonstrates its effectiveness at sound race detection on a suite of seven multithreaded Java programs.

3.1 Introduction

Most approaches to proving race freedom for shared-memory multithreaded programs focus on checking lock-based synchronization [11, 12, 28, 29, 39, 69, 73]. This style of synchronization requires that any pair of otherwise unordered accesses from different threads to the same memory location m must be *guarded* by a lock l in that each thread must hold lock l while accessing m . Since at most one thread can hold lock l at any instant, there are no races on m if lock-based synchronization is used correctly.

A challenge in proving race freedom in the presence of locks lies in the apparent

need for a form of must alias analysis. Consider the following pseudo-code example:

```

// Thread t1 executes:           // Thread t2 executes:
    synchronized (l1) {           synchronized (l2) {
s1:      e1.f = ...;              s2:      e2.f = ...;
    }                             }

```

Here, “`synchronized (l) { s }`” is Java’s lexically-scoped locking construct: the thread executing it acquires a lock on the object denoted by `l` before executing `s` and releases the lock upon finishing executing `s`. The statements labeled `s1` and `s2` write to memory locations denoted by instance field `f` of the objects denoted by expressions `e1` and `e2`, respectively. Suppose it is possible that `e1 = e2` in some execution, that is, `e1` and `e2` *may alias*. Then, to prove that `s1` and `s2` are race-free, it suffices to prove that `l1 = l2` in every execution, that is, `l1` and `l2` *must alias*.

Must alias analysis is perceived as a harder problem than may alias analysis, and the literature on must alias analysis, unlike that on may alias analysis, is very small. Hence, the apparent need for a must alias analysis to prove that a pair of statements is ordered by lock-based synchronization has been a major impediment to many previous static race detection approaches. Indeed, folk wisdom that static race detection is intractable is primarily attributed to this problem.

In Chapter 2, we presented a static race detection algorithm for Java that consists of four static analyses each of which checks a different condition for race freedom while together enabling the algorithm to output a useful set of potential races. We used a lockset analysis for checking the condition that a pair of statements is race-free if it is ordered by lock-based synchronization. Our lockset analysis, however, is unsound as it is based upon a may alias analysis instead of a must alias analysis. In particular, for the above example, it infers that a common lock *may* be held by threads `t1` and `t2` while executing statements `s1` and `s2`, respectively, if `l1` and `l2` may alias, that is, if the intersection of the points-to sets of `l1` and `l2` is non-empty.

Some static race detection approaches based on alias analysis (e.g., [17, 18]) check whether `l1` and `l2` must alias by checking whether the points-to sets of `l1` and `l2` are equal and contain a single abstract object and, furthermore, at most one concrete

object is abstracted by that lone abstract object in any execution. For instance, if the abstract object is an object allocation site, then it is sufficient to prove that the site is executed at most once in any execution. Such approaches are sound but they are suitable only for programs with coarse-grained parallelism which use global, uniquely named locks; they are ineffective on programs with fine-grained parallelism which create multiple locks at run-time that are stored in data structures and passed around by functions.

In this chapter, we present a novel approach to sound race detection in the presence of locks. The key idea is that instead of attacking the problem directly using a must alias analysis, we reformulate it as a dual *must not alias* analysis problem. Consider the above example once again. Instead of starting with accessed objects **e1** and **e2** and reasoning about lock objects **l1** and **l2**, we start with the lock objects and try to reason about the accessed objects. In particular, if under the assumption that objects **l1** and **l2** are not the same (the lock objects must not alias), we can prove that objects **e1** and **e2** are not the same (the accessed objects must not alias), then **s1** and **s2** are race-free. Intuitively, if whenever two locks are different, the memory locations they guard are also different, then there are no races. This approach to proving race freedom is an application of a property we call *conditional must not aliasing*: Under the assumption that two objects are not aliased, prove that two other objects are not aliased. Note that this property also handles the case of global, uniquely named locks: in the above example, if **l1** and **l2** must alias, the antecedent of the conditional must not aliasing property is false and hence the property is trivially true, from which it follows that **s1** and **s2** are race-free.

We have devised a sound conditional must not alias analysis that replaces the unsound lockset analysis in our static race detection algorithm. The analysis is based on a *disjoint reachability analysis* that uses a type and effect system to construct an abstraction of how objects are created and linked in the heap. The disjoint reachability analysis answers queries posed by the conditional must not alias analysis regarding reachability in the heap between lock objects such as **l1** and **l2** and accessed objects such as **e1** and **e2** in the above example.

We have implemented our static race detection algorithm using conditional must

not alias analysis and applied it to a suite of seven multithreaded Java programs. Our experiments demonstrate that the approach has a false positive rate of 25%, that is, only one in every four reported races is a non-race. Also, because the approach is sound for complete programs (namely, programs that do not have missing callers or callees and that do not use dynamic class loading and reflection), it does not have false negatives.

The rest of this chapter is organized as follows. Section 3.2 explains the idea of conditional must not aliasing for proving race freedom for an example Java program. Section 3.3 presents a WHILE language, Section 3.4 presents a type and effect system for the language, and Section 3.5 presents our disjoint reachability analysis built upon the type and effect system. Section 3.6 discusses features that are elided in the WHILE language but are necessary for handling realistic Java programs. Section 3.7 presents our conditional must not alias analysis in the context of our static race detection algorithm. Section 3.8 shows the effectiveness of the analysis at performing sound race detection on a suite of seven multithreaded Java programs. Finally, Section 3.9 surveys related work.

3.2 Example

In this section, we elucidate the idea of conditional must not aliasing in the context of proving race freedom for an example multithreaded Java program using various locking idioms. The program is shown in Figure 3.1. It begins by executing the `main` method of class `T` in an implicit main thread. The main thread creates an array of `T` objects at object allocation site labeled `h1` and stores the array in static field `g` of class `T`, which may be viewed as a global variable. It then creates a bunch of `T` objects in a loop. While constructing each `T` object, which is allocated at the site labeled `h2`, instance field `f1` of that `T` object is assigned a fresh `C` object, which is allocated at the site labeled `h3`. Each `C` object has an integer field `f2`. The resulting data structure is shown in Figure 3.2. It uses (i, h) to denote the object allocated at site labeled h in the i^{th} iteration of the loop; $i = 0$ means the object was allocated outside the loop.

Immediately after creating a `T` object in a particular iteration of the loop, the main

```

public class T extends java.lang.Thread {
    public static void main(String[] a) {
h1:      T.g = new T[*];
          for (int i = 0; i < *; i++) {
h2:      T vt = new T();
          T.g[i] = vt;
          vt.start();
        }
    }
    private static T[] g;
    private C f1;
    public T() {
h3:      this.f1 = new C();
    }
    public void run() {
        T vt = thrd;          // Choices for thrd: T.g[*], this
        C ve = vt.f1;
        ? vl = lock;          // Choices for lock: T.g, vt, ve
        synchronized (vl) {
e:        ve.f2 = ...;
        }
    }
}

public class C {
    public int f2;
}

```

Figure 3.1: Example multithreaded Java program.

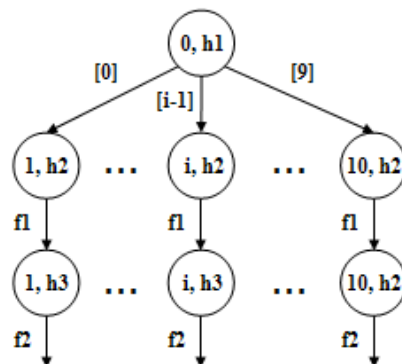


Figure 3.2: Data structure created by example program.

thread calls the `start` method of superclass `java.lang.Thread` on that `T` object. The `start` method, which is not shown, invokes the `run` method of class `T` on that `T` object in a fresh child thread. The calls to the `start` method are asynchronous: the main thread continues executing the `main` method while each of the previously spawned child threads executes the `run` method.

Each child thread executing the `run` method first assigns the object denoted by expression `thrd` to local variable `vt`. We consider two scenarios. In the first scenario, expression `thrd` is `T.g[*]`, that is, each child thread non-deterministically chooses a `T` object corresponding to one of the child threads. Note that since the choice is non-deterministic, one child thread may choose a `T` object corresponding to a child thread other than itself. In the second scenario, the expression `thrd` is `this`, that is, each child thread chooses the `T` object corresponding to itself.

Each child thread then follows field `f1` of its chosen `T` object to access the corresponding `C` object, and stores the `C` object in local variable `ve`. Finally, the child thread assigns the object denoted by expression `lock` to local variable `v1`, acquires a lock on that object, writes to the memory location denoted by field `f2` of the `C` object it stored in local variable `ve`, and releases the lock.

Note that in the scenario in which expression `thrd` is `T.g[*]`, this program has a potential race: two different child threads can non-deterministically choose the same `T` object, follow its `f1` field to access the same `C` object and, if the lock next acquired by the two threads is not the same, then they can write to the same memory location denoted by field `f2` of the accessed `C` object. We next illustrate three different choices for expression `lock`, so far left unspecified, that prevent this race.

Consider the case where expression `lock` is `T.g`, that is, each child thread holds a lock on the array of `T` objects. This case represents a coarse-grained locking style in which global, uniquely named locks are used, namely, each such lock is created at an object allocation site that is executed exactly once. For instance, the array of `T` objects on which the lock is held by each child thread is allocated at site `h1`, which is executed exactly once. Some static race detection approaches based on alias analysis (e.g., [17, 18]) rely on such object allocation sites which are executed exactly once for proving race freedom. From the point of view of conditional must not aliasing, this

case is uncomplicated. Recall that under the assumption that two locks are different, if we can prove that the memory locations they guard are also different, then the conditional must not aliasing property holds and there are no races. Consider any two different child threads. Since the assumption that the locks they hold are different is false (they always hold the same lock on the array of T objects), the antecedent of the property is false and hence the property is trivially true and the program is race-free.

Now consider the case where expression `lock` is `ve`, that is, each child thread holds a lock on the C object to whose field `f2` it writes. This case represents the extreme of fine-grained locking, and once again, reasoning using conditional must not aliasing to prove race freedom is straightforward. Consider any two different child threads. Assume that they hold locks on different C objects. Then, the `f2` fields of those C objects to which the threads write are also different memory locations (fields of distinct objects in Java have distinct memory locations), and hence the conditional must not aliasing property holds and the program is race-free.

Now consider the case where expression `lock` is `vt`, that is, each child thread holds a lock on the T object it chose non-deterministically, and follows its `f1` field to access the C object to whose field `f2` it writes. We call this case “medium-grained locking”, and once again, we can show using conditional must not aliasing that the program is race-free, though in this case the reasoning is subtle. Consider any two different child threads. Assume that they hold locks on different T objects. Then, the `f2` fields of the C objects to which the threads write are different memory locations because the `f1` fields of different T objects *always* point to different C objects, and hence the program is race-free. Automatically inferring this fact, however, requires reasoning about different objects allocated at the same syntactic site in the program, such as different T objects allocated at site `h2` and different C objects allocated at site `h3`, and how these objects are linked in pointer-based data structures. We employ a *disjoint reachability analysis* that uses a type and effect system to build an abstraction of how objects are created and linked in the heap. The disjoint reachability analysis answers conditional must not aliasing queries regarding reachability in the heap between lock objects and accessed objects. In the above example, for instance, it answers that the

same **C** object allocated at site **h3** is not reachable in the heap from different **T** objects allocated at site **h2** or, equivalently, from different **T** objects allocated at site **h2** only different **C** objects allocated at site **h1** may be reachable in the heap (hence the name “disjoint reachability”).

Each of the above three locking styles is fairly common in realistic Java programs. Note that we have so far considered the scenario in which expression `thrd` is `T.g[*]`. This scenario exhibits a style in which a group of child threads operating on a data structure use lock-based synchronization to prevent races whenever they access shared parts of the structure. For the other scenario, in which expression `thrd` is `this`, the locking is redundant because each child thread deterministically chooses the **T** object corresponding to itself, and so any two different child threads always choose different **T** objects, follow their **f1** fields to access different **C** objects, and then write to their **f2** fields which are different memory locations. This scenario is also fairly common in real-world Java programs and exhibits a style in which a group of child threads operate on disjoint parts of a data structure, and the main thread consolidates the result after they are done. Note, however, that although locking is not necessary to prevent races in this scenario, a conditional must not aliasing style of reasoning is still required to prove race freedom. The only difference is that instead of lock objects we now have thread objects, namely, under the assumption that two threads are different, if we can prove that the memory locations they access are also different, then the program is race-free.

3.3 Language

In this section, we present the abstract syntax and operational semantics of a sequential, intraprocedural WHILE language that we use in subsequent sections to formalize our conditional must not alias analysis.

3.3.1 Syntax

The abstract syntax of the language is shown in Figure 3.3. A program has a fixed set of variables $\mathbb{V}$ with global scope and a single class with instance fields $\mathbb{F}$. Each object allocation site in the program is labeled with a unique $h \in \mathbb{H}$ and each loop in the program is labeled with a unique integer $w \in \mathbb{W}$. There are no threads or locks; conditional must not aliasing is not a concurrency property and the presentation is simplest in a single-threaded language.

v	$\in$	$\mathbb{V}$	(variable)
f	$\in$	$\mathbb{F}$	(instance field)
h	$\in$	$\mathbb{H}$	(object allocation site)
w	$\in$	$\mathbb{W}$	(loop)
s	$\in$	$\mathbb{S}$	(statement)
s	$::=$	$v = \text{null}$	
		$v = \text{new } h$	
		$v_1 = v_2$	
		$v_2 = v_1.f$	
		$v_1.f = v_2$	
		$s_1 ; s_2$	
		if $(*)$ then s_1 else s_2	
		while ^{w} $(*)$ do s	

Figure 3.3: Abstract syntax of WHILE language.

3.3.2 Semantics

We next develop an operational semantics for the language. Figure 3.4 defines the semantic domains. A *loop vector* π is a tuple of $|\mathbb{W}|$ non-negative integers which track how many times each loop in the program has executed. Specifically, the iteration count of a loop **while** ^{w} $(*)$ **do** s in the program is $\pi(w)$ (treating the tuple π as a map from indices to elements of π). A (non-null) object o , then, is uniquely identified as a pair $\langle h, \pi \rangle$ consisting of the site h at which it was allocated and a loop vector π recording the time (in loop execution counts) when the object was allocated. We explain the motivation behind using loop vectors shortly.

$\mathbb{N}$	$\ni$	n	$::=$	$0 \mid 1 \mid 2 \mid \dots$	(loop iteration count)
		π	$\in$	$\mathbb{W} \rightarrow \mathbb{N}$	(loop iteration count vector)
		W	$\in$	$\mathcal{P}(\mathbb{W})$	(loop set)
$\mathbb{O}$	$\ni$	o	$::=$	$\langle h, \pi \rangle$	(non-null object)
$\mathbb{O}_\perp$	$\ni$	$\bar{o}$	$::=$	$o \mid \perp$	(object)
		ρ	$\in$	$\mathbb{V} \rightarrow \mathbb{O}_\perp$	(environment)
		σ	$\in$	$(\mathbb{O} \times \mathbb{F}) \rightarrow \mathbb{O}_\perp$	(heap)
		C	$::=$	$\emptyset \mid C \cup \{ o_1 \triangleright o_2 \}$	(heap effect set)

$$\begin{aligned} \langle h, \pi \rangle . \mathbf{h} &\triangleq h \\ \langle h, \pi \rangle . \mathbf{\pi} &\triangleq \pi \end{aligned}$$

Figure 3.4: Semantic domains of WHILE language.

Environments and heaps are standard. An environment ρ maps each variable to an object (or null). A heap σ records the object (or null) to which each field of each non-null object points. A *heap effect* $o_1 \triangleright o_2$ records that at some point in the execution, some field of object o_1 was assigned object o_2 . The field itself is not recorded as we are only interested in object reachability, namely, that object o_2 was reachable by one field dereference from object o_1 .

The motivation behind loop vectors and heap effects is that in Section 3.4, we will present a type and effect system that is used by a disjoint reachability analysis to answer conditional must not aliasing queries such as whether the same object allocated at a site in the program may be reachable in the heap from different objects allocated at another site. The type and effect system uses abstractions of the loop vectors to reason about different objects allocated at the same site and abstractions of the heap effects to reason about reachability between them in the heap.

Figure 3.5 presents a big-step operational semantics for our WHILE language. Judgments have the form:

$$s, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C$$

Each step of execution begins with the statement s to be executed, the set W of all loops lexically enclosing s , the current loop vector π , the current environment ρ , and

$$\boxed{s, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C}$$

$$v = \text{null}, W, \pi, \rho, \sigma \Downarrow \pi, \rho[v \mapsto \perp], \sigma, \emptyset \quad (3.1)$$

$$v = \text{new } h, W, \pi, \rho, \sigma \Downarrow \pi, \rho[v \mapsto o], \sigma[(o, f_1) \mapsto \perp, \dots, (o, f_n) \mapsto \perp], \emptyset \\ [o = \langle h, \lambda w. (\text{if } w \in W \text{ then } \pi(w) \text{ else } 0) \rangle] \quad (3.2)$$

$$v_1 = v_2, W, \pi, \rho, \sigma \Downarrow \pi, \rho[v_1 \mapsto \rho(v_2)], \sigma, \emptyset \quad (3.3)$$

$$v_2 = v_1.f, W, \pi, \rho, \sigma \Downarrow \pi, \rho[v_2 \mapsto \sigma(o, f)], \sigma, \emptyset \quad \text{if } \rho(v_1) = o \quad (3.4)$$

$$v_1.f = v_2, W, \pi, \rho, \sigma \Downarrow \pi, \rho, \sigma[(o_1, f) \mapsto \bar{o}_2], C \quad \text{if } \rho(v_1) = o_1 \\ \left[\rho(v_2) = \bar{o}_2 \text{ and } C = \begin{cases} \{o_1 \triangleright o_2\} & \text{if } \bar{o}_2 = o_2 \\ \emptyset & \text{if } \bar{o}_2 = \perp \end{cases} \right] \quad (3.5)$$

$$\frac{s_1, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C_1 \quad s_2, W, \pi', \rho', \sigma' \Downarrow \pi'', \rho'', \sigma'', C_2}{s_1; s_2, W, \pi, \rho, \sigma \Downarrow \pi'', \rho'', \sigma'', C_1 \cup C_2} \quad (3.6)$$

$$\frac{s_1, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C}{\text{if } (*) s_1 \text{ else } s_2, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C} \quad (3.7)$$

$$\frac{s_2, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C}{\text{if } (*) s_1 \text{ else } s_2, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C} \quad (3.8)$$

$$\text{while}^w (*) \text{ do } s, W, \pi, \rho, \sigma \Downarrow \pi, \rho, \sigma, \emptyset \quad (3.9)$$

$$\frac{s, W \cup \{w\}, \pi[w \mapsto \pi(w) + 1], \rho, \sigma \Downarrow \pi', \rho', \sigma', C_1 \quad \text{while}^w (*) \text{ do } s, W, \pi', \rho', \sigma' \Downarrow \pi'', \rho'', \sigma'', C_2}{\text{while}^w (*) \text{ do } s, W, \pi, \rho, \sigma \Downarrow \pi'', \rho'', \sigma'', C_1 \cup C_2} \quad (3.10)$$

Figure 3.5: Instrumented operational semantics of WHILE language.

the current heap σ . Note that W records which loops are currently executing while π records the execution count of all loops in the program and not just of loops that are currently active. Since loops may execute as part of a step of execution, the semantics must record a new loop vector as well as an updated environment and heap. Thus, a step of execution terminates with a final loop vector π' , a final environment ρ' , and a final heap σ' , plus the set of heap effects C recording all object reachability facts resulting from heap updates that occurred during the execution of s .

We explain the most interesting rules in Figure 3.5. Rule (3.2), which creates a new object, does not simply use the current loop vector as the time stamp recorded in the object. Instead, counters for loops not in W (that is, those not currently executing) are set to 0, giving a way to determine later whether or not a particular loop was executing when the object was allocated.¹ While this property is not exploited in the instrumented operational semantics, it is used in the abstractions discussed in Section 3.4. Assigning to a field of an object (Rule (3.5)) generates a heap effect recording the reachability between the two objects involved in the assignment. Finally, consider Rules (3.9) and (3.10), which give the semantics of **while** statements. Loops execute a non-deterministic number of times, which saves us the trouble of defining how loop termination conditions (as well as the predicates of **if** statements, see Rules (3.7) and (3.8)) are evaluated. Also note that when a loop executes an additional time (Rule (3.10)), the appropriate loop counter in the current loop vector is incremented.

We conclude this section with a small example in our WHILE language:

Example 3.1. $\text{while}^1 (*) \text{ do } \{ v_1 = \text{new } h_1; v_2 = \text{new } h_2; v_1.f = v_2 \}$

If the loop executes twice, the judgment at the root of the derivation tree is:

$$\text{while}^1 \dots, \emptyset, \langle 0 \rangle, [v_1 \mapsto \perp, v_2 \mapsto \perp], [] \Downarrow \langle 2 \rangle, [v_1 \mapsto o_3, v_2 \mapsto o_4], \sigma, \{o_1 \triangleright o_2, o_3 \triangleright o_4\}$$

where $o_1 = \langle h_1, \langle 1 \rangle \rangle$, $o_2 = \langle h_2, \langle 1 \rangle \rangle$, $o_3 = \langle h_1, \langle 2 \rangle \rangle$, $o_4 = \langle h_2, \langle 2 \rangle \rangle$, the initial empty heap is denoted $[]$, and the final heap is $\sigma = [(o_1, f) \mapsto o_2, (o_2, f) \mapsto \perp, (o_3, f) \mapsto o_4, (o_4, f) \mapsto \perp]$.

¹The object allocation site could also be used to determine the set of lexically enclosing loops; using W is clearer if less economical.

3.4 Type and Effect System

In this section, we present a type and effect system for our WHILE language. The syntax of types and effects is shown in Figure 3.6. They are parallel with the definitions in Figure 3.4 but the semantics are significantly different. Objects have types $\langle \hat{h}, \Pi \rangle$ recording information about where and when they were allocated. The main goal of the type and effect system is to compute abstract heap effects such as $\langle \hat{h}_1, \Pi_1 \rangle \supseteq \langle \hat{h}_2, \Pi_2 \rangle$. As in the operational semantics, the effect implies an object of type $\langle \hat{h}_2, \Pi_2 \rangle$ may be reachable from an object of type $\langle \hat{h}_1, \Pi_1 \rangle$ by one field dereference. In a type, loop iterations are abstracted as 0, 1, or $\top$. If $\Pi_1(w) = 0$, then loop w was not executing when the object with that type was allocated (similarly for $\Pi_2(w)$). If $\Pi_1(w) = \top$ then nothing is known about the iteration of loop w in which the object was allocated (and similarly for $\Pi_2(w)$). In either case nothing is known about the relative time at which objects of the two types were allocated. However, if $\Pi_1(w) = \Pi_2(w) = 1$, then the type and effect system ensures the two objects were allocated in the *same* iteration of loop w . This property allows us to show disjoint reachability: intuitively, if objects of types $\langle \hat{h}_1, \Pi_1 \rangle$ and $\langle \hat{h}_2, \Pi_2 \rangle$ are allocated and linked in the same iteration of a loop, then different objects of type $\langle \hat{h}_1, \Pi_1 \rangle$ (allocated in different iterations n_1 and n_2 of the loop) may only reach different objects of type $\langle \hat{h}_2, \Pi_2 \rangle$ (allocated in iterations n_1 and n_2 , respectively).

This discussion is made precise in Figure 3.7, which defines an abstraction relation $\preceq$ stating when types, abstract heap effects, and type environments abstract objects, concrete heap effects, and concrete environments, respectively. The third clause in Figure 3.7(b) requires that the iteration counts in position w of the loop vectors of two objects match if the values in position w of the loop vectors of their types are 1. Likewise, the third clause of Figure 3.7(c) requires that iteration counts in position w of the loop vectors of all objects in environment ρ match if the values in position w of the loop vectors of their types in environment Γ are 1. Thus, in both abstract heap effects and type environments, any two types with a 1 in position w of their loop vectors always abstract objects allocated in the same, but unknown, iteration of loop w .

$\mathbb{N}_\top$	$\ni \hat{n} ::= 0 \mid 1 \mid \top$	(abstract loop iteration count)
	$\Pi \in \mathbb{W} \rightarrow \mathbb{N}_\top$	(abstract loop iteration count vector)
$\mathbb{H}_\top$	$\ni \hat{h} ::= h \mid \top$	(abstract object allocation site)
$\mathbb{T}$	$\ni \tau ::= \langle \hat{h}, \Pi \rangle$	(non-null type)
$\mathbb{T}_\perp$	$\ni \bar{\tau} ::= \tau \mid \perp$	(type)
	$\Gamma \in \mathbb{V} \rightarrow \mathbb{T}_\perp$	(type environment)
	$K ::= \emptyset \mid K \cup \{\tau_1 \sqsupseteq \tau_2\}$	(abstract heap effect set)

$$\begin{aligned} \langle \hat{h}, \Pi \rangle . \hat{\mathbf{h}} &\triangleq \hat{h} \\ \langle \hat{h}, \Pi \rangle . \mathbf{\Pi} &\triangleq \Pi \end{aligned}$$

Figure 3.6: Syntax of types and effects.

Before we can give the type rules we need two operations on type environments. The join of type environments is pointwise. Nulls are absorbed, and if either loop iterations or object allocation sites fail to match, the result is $\top$ in the appropriate position. The second operation handles the increment of loop vectors; in a $\text{while}^w(*)$ **do** s statement, if the value in position w of the loop vector is 1, it is incremented to $\top$ when the loop iterates.

$$\begin{aligned}
\bar{o} \preceq \bar{\tau} &\Leftrightarrow (\bar{o} = \perp) \vee (\bar{o} = o \wedge \bar{\tau} = \tau \wedge o \preceq \tau) \\
o \preceq \tau &\Leftrightarrow (o.\mathbf{h} \preceq \tau.\hat{\mathbf{h}}) \wedge (\forall w \in \mathbb{W} : o.\boldsymbol{\pi}(w) \preceq \tau.\mathbf{\Pi}(w)) \\
n \preceq \hat{n} &\Leftrightarrow (n = 0 \wedge \hat{n} = 0) \vee (n > 0 \wedge \hat{n} = 1) \vee (\hat{n} = \top) \\
h \preceq \hat{h} &\Leftrightarrow (h = \hat{h}) \vee (\hat{h} = \top)
\end{aligned}$$

(a) Object abstraction.

$$\begin{aligned}
C \preceq K &\Leftrightarrow \forall (o_1 \triangleright o_2) \in C : \exists (\tau_1 \triangleright \tau_2) \in K : (o_1, o_2) \propto (\tau_1, \tau_2) \\
(o_1, o_2) \propto (\tau_1, \tau_2) &\Leftrightarrow \left(\begin{array}{l} (1) o_1 \preceq \tau_1 \\ \wedge (2) o_2 \preceq \tau_2 \\ \wedge (3) \forall w \in \mathbb{W} : ((\tau_1.\mathbf{\Pi}(w) = 1 \wedge \tau_2.\mathbf{\Pi}(w) = 1) \Rightarrow \\ \quad o_1.\boldsymbol{\pi}(w) = o_2.\boldsymbol{\pi}(w)) \end{array} \right)
\end{aligned}$$

(b) Heap effect abstraction.

$$W \vdash (\pi, \rho) \preceq (\Pi, \Gamma) \Leftrightarrow \left(\begin{array}{l} (1) \forall w \in W : \pi(w) \preceq \Pi(w) \\ \wedge (2) \forall v \in \mathbb{V} : \rho(v) \preceq \Gamma(v) \\ \wedge (3) \forall w \in \mathbb{W} : \exists n \in \mathbb{N} : \\ \quad \left[\begin{array}{l} (a) \Pi(w) = 1 \Rightarrow \pi(w) = n \\ \wedge (b) \forall v \in \mathbb{V} : ((\rho(v) = o \wedge \Gamma(v) = \tau \wedge \\ \quad \tau.\mathbf{\Pi}(w) = 1) \Rightarrow o.\boldsymbol{\pi}(w) = n) \end{array} \right] \end{array} \right)$$

(c) Environment abstraction.

Figure 3.7: Abstraction relations.

Definition 3.2. (Join of Environments)

$$\begin{aligned}
(\Gamma_1 \sqcup \Gamma_2)(v) &= \Gamma_1(v) \sqcup \Gamma_2(v) \\
\bar{\tau}_1 \sqcup \bar{\tau}_2 &= \begin{cases} \bar{\tau}_1 & \text{if } \bar{\tau}_2 = \perp \\ \bar{\tau}_2 & \text{if } \bar{\tau}_1 = \perp \\ \tau_1 \sqcup \tau_2 & \text{if } \bar{\tau}_1 = \tau_1 \wedge \bar{\tau}_2 = \tau_2 \end{cases} \\
\tau_1 \sqcup \tau_2 &= \langle \tau_1.\hat{\mathbf{h}} \sqcup \tau_2.\hat{\mathbf{h}}, \tau_1.\mathbf{\Pi} \sqcup \tau_2.\mathbf{\Pi} \rangle \\
\hat{h}_1 \sqcup \hat{h}_2 &= \begin{cases} \hat{h}_1 & \text{if } \hat{h}_1 = \hat{h}_2 \\ \top & \text{otherwise} \end{cases} \\
(\Pi_1 \sqcup \Pi_2)(w) &= \Pi_1(w) \sqcup \Pi_2(w) \\
\hat{n}_1 \sqcup \hat{n}_2 &= \begin{cases} \hat{n}_1 & \text{if } \hat{n}_1 = \hat{n}_2 \\ \top & \text{otherwise} \end{cases}
\end{aligned}$$

□

Definition 3.3. (Loopback Environment)

$$\begin{aligned}
\Gamma^{w+}(v) &= \Gamma(v)^{w+} \\
\tau^{w+} &= \langle \tau.\hat{\mathbf{h}}, \tau.\mathbf{\Pi}^{w+} \rangle \\
\perp^{w+} &= \perp \\
\Pi^{w+}(w') &= \begin{cases} \top & \text{if } w' = w \wedge \Pi(w) = 1 \\ \Pi(w') & \text{otherwise} \end{cases}
\end{aligned}$$

□

The upper bound of two types implicitly defines a type lattice, which is ordered pointwise on loop vectors and object allocation sites. Integers and object allocation sites are all less than $\top$ and incomparable to each other. The maximal type is then $\langle \top, \lambda w. \top \rangle$, that is, it consists of a $\top$ object allocation site and a loop vector with a $\top$ in each position; the minimal type is the type $\perp$ of null, and since any program has a finite number of loops and object allocation sites, the type lattice is also finite.

$$\boxed{W, \Pi, \Gamma \vdash s : \Gamma', K}$$

$$W, \Pi, \Gamma \vdash v = \mathbf{null} : \Gamma[v \mapsto \perp], \emptyset \quad (3.11)$$

$$\begin{aligned} W, \Pi, \Gamma \vdash v = \mathbf{new} \ h : \Gamma[v \mapsto \langle h, \Pi' \rangle], \emptyset \\ [\Pi' = \lambda w. (\text{if } w \in W \text{ then } \Pi(w) \text{ else } 0)] \end{aligned} \quad (3.12)$$

$$W, \Pi, \Gamma \vdash v_1 = v_2 : \Gamma[v_1 \mapsto \Gamma(v_2)], \emptyset \quad (3.13)$$

$$W, \Pi, \Gamma \vdash v_2 = v_1.f : \Gamma[v_2 \mapsto \langle \top, \lambda w. \top \rangle], \emptyset \quad (3.14)$$

$$\begin{aligned} & W, \Pi, \Gamma \vdash v_1.f = v_2 : \Gamma, K \\ & \left[K = \begin{cases} \{\tau_1 \sqsupseteq \tau_2\} & \text{if } \Gamma(v_1) = \tau_1 \text{ and } \Gamma(v_2) = \tau_2 \\ \emptyset & \text{otherwise} \end{cases} \right] \end{aligned} \quad (3.15)$$

$$\frac{W, \Pi, \Gamma \vdash s_1 : \Gamma', K_1 \quad W, \Pi, \Gamma' \vdash s_2 : \Gamma'', K_2}{W, \Pi, \Gamma \vdash s_1; s_2 : \Gamma'', K_1 \cup K_2} \quad (3.16)$$

$$\frac{W, \Pi, \Gamma \vdash s_1 : \Gamma_1, K_1 \quad W, \Pi, \Gamma \vdash s_2 : \Gamma_2, K_2}{W, \Pi, \Gamma \vdash \text{if } (*) \text{ then } s_1 \text{ else } s_2 : \Gamma_1 \sqcup \Gamma_2, K_1 \cup K_2} \quad (3.17)$$

$$\frac{W \cup \{w\}, \Pi, \Gamma^{w+} \vdash s : \Gamma, K}{W, \Pi, \Gamma \vdash \mathbf{while}^w (*) \text{ do } s : \Gamma, K} \quad [\Pi(w) \neq 0] \quad (3.18)$$

Figure 3.8: Type rules.

The type rules are given in Figure 3.8. A type judgment has the form:

$$\mathbb{W}, \Pi, \Gamma \vdash s : \Gamma', K$$

$\mathbb{W}$ is the set of all loops lexically enclosing statement s . The type system is flow sensitive: if statement s begins executing with initial loop vector Π and initial type environment Γ , then upon termination the final type environment is Γ' . Note that unlike in the judgment for the operational semantics, the final loop vector is the same as the initial loop vector; the type rules ensure that any conclusion that is drawn holds for any number of iterations of loops executed during the execution of s . Finally, K denotes the set of abstract heap effects that may occur during the execution of s .

The type rules are parallel with the operational semantics in Figure 3.5 and for brevity we point out only a few interesting features. Rule (3.12) uses $\mathbb{W}$ to distinguish active loops from inactive ones. The type of a newly allocated object has 0's in loop vector positions of inactive loops, just as in Rule (3.2) of Figure 3.5, and the values for loop vector positions of active loops are taken from the current loop vector Π . Rule (3.14) gives no information about heap reads, which is sound, but overly conservative in practice. We discuss improvements in Section 3.8, which we omit from the formal development for simplicity. The most interesting rule, Rule (3.18), has three important aspects. First, the condition $\Pi(w) \neq 0$ reflects that the loop vectors of the types of objects allocated inside loop w should not have 0 at position w (and Rule (3.12) already ensures that the loop vectors of the types of objects allocated outside loop w have a 0 at position w). Second, the fact that the environment Γ is the same before and after the loop reflects that any conclusion must be valid for any number of executions of the loop, that is, the entire loop may be executed multiple times (e.g., because it is nested inside of another loop) and the environment Γ must be an invariant for all of those executions. For example, a proof $\mathbb{W}, \Pi, \Gamma \vdash \text{while}^w \dots : \Gamma, K$ where $\Gamma = [v_1 \mapsto \langle h_1, \langle \dots, 1, \dots \rangle \rangle, v_2 \mapsto \langle h_2, \langle \dots, 1, \dots \rangle \rangle]$ implies that if the loop starts execution in an environment where v_1 and v_2 were allocated in the same iteration of some earlier execution of the loop (e.g., because it is nested inside of another loop), then the loop terminates with v_1 and v_2 assigned objects from the same loop iteration.

Note that the final concrete loop iteration associated with v_1 and v_2 may be different than the initial one; the value 1 in both types only requires that the concrete loop iterations of v_1 and v_2 be equal before and after the loop, but the loop may assign new objects to v_1 and v_2 from the same iteration and maintain this property. Third and finally, the types of objects in the environment at the start of a loop iteration must be carried over from previous iterations. Thus, the body s of loop w is checked in the environment Γ^{w+} , which ensures that the types of objects in the environment at the start of a new iteration do not have a 1 in position w of their loop vectors; $\Pi(w)$, however, can be 1, which allows the types of any objects s allocates to be recognized as the types of objects allocated together in the same iteration.

As an aside, for a single loop the only correlation this type and effect system can recognize is when objects are allocated and linked in the same iteration of the loop (Rule (3.15)). By allowing loop vector positions of types to take a finite number of additional values (that is, values 2,3,4, ... besides the currently allowed values 0,1, and $\top$) and adjusting definitions (e.g., Definition 3.3) the system can be extended to recognize when a value is allocated in one iteration and linked to an object allocated in the next iteration, or two iterations later, and so on. However, so far we have not found this extra power necessary, at least for race detection, and so we have presented and implemented the simpler system. Much more important is correctly handling non-nested and multiple nested loops and this is the focus of our system.

The purpose of the type and effect system is to compute the set of abstract heap effects which is used by our disjoint reachability analysis presented in the following section. We first prove the soundness of abstract heap effects with respect to concrete heap effects. The key lemma in this proof is type preservation.

Lemma 3.4. (Type Preservation) *If $s, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C$ and $W, \Pi, \Gamma \vdash s : \Gamma', K$ and $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ then $W \vdash (\pi', \rho') \preceq (\Pi, \Gamma')$ and $C \preceq K$.*

Proof. By induction on the structure of the derivation of $s, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C$. See the proof of Lemma A.2 in Appendix A for details. $\square$

Then, the following theorem proves the soundness of the set of abstract heap effects K for a well-typed program with respect to the set of concrete heap effects C of any

execution of that program.

Theorem 3.5. (Soundness of Heap Effect Abstraction)

If $s, \emptyset, \lambda w.0, \lambda v.\perp, [] \Downarrow \pi, \rho, \sigma, C$ and $\emptyset, \Pi, \Gamma \vdash s : \Gamma', K$ then $C \preceq K$.

Proof. Suppose:

(a) $s, \emptyset, \lambda w.0, \lambda v.\perp, [] \Downarrow \pi, \rho, \sigma, C$

(b) $\emptyset, \Pi, \Gamma \vdash s : \Gamma', K$

To prove:

(I) $C \preceq K$

From Figure 3.7 (c), we have:

(c) $\emptyset \vdash (\lambda w.0, \lambda v.\perp) \preceq (\Pi, \Gamma)$

From (a), (b), (c), and Lemma 3.4, we have (I). □

We will use this theorem to prove the soundness of our disjoint reachability analysis with respect to the disjoint reachability property in the following section (see Theorem 3.13).

Returning to Example 3.1 at the end of Section 3.3, the type system can prove:

$$\emptyset, \langle 1 \rangle, [v_1 \mapsto \tau_1, v_2 \mapsto \tau_2] \vdash \text{while}^1 \dots : [v_1 \mapsto \tau_1, v_2 \mapsto \tau_2], \{\tau_1 \supseteq \tau_2\}$$

where $\tau_1 = \langle h_1, \langle 1 \rangle \rangle$ and $\tau_2 = \langle h_2, \langle 1 \rangle \rangle$.

Example 3.6. Consider the following nested loop, with two possible statements (A) and (B) for the body of the inner loop:

```
while1 (*) do
  v1 = new h1;
  while2 (*) do
    v2 = new h2;
    (A) v1.f = v2 OR (B) v2.f = v1
```

Statement (A) abstracts a typical programming pattern for containers: the outer object v_1 controls access to objects v_2 allocated in an inner loop (in realistic examples

$$\begin{aligned}
h \in DR_C(H) &\Leftrightarrow \left(\left[\begin{array}{l} (o_1 \triangleright o) \in C^+ \\ \wedge \quad (o_2 \triangleright o) \in C^+ \\ \wedge \quad o_1.\mathbf{h} \in H \\ \wedge \quad o_2.\mathbf{h} \in H \\ \wedge \quad o.\mathbf{h} = h \end{array} \right] \Rightarrow o_1 = o_2 \right) \\
h \in DR_K(H) &\Leftrightarrow \left(\left[\begin{array}{l} (\tau_1 \supseteq \tau_3) \in K^+ \\ \wedge \quad (\tau_2 \supseteq \tau_4) \in K^+ \\ \wedge \quad \tau_1.\hat{\mathbf{h}} \in H \cup \{\top\} \\ \wedge \quad \tau_2.\hat{\mathbf{h}} \in H \cup \{\top\} \\ \wedge \quad \tau_3 \sim \tau_4 \\ \wedge \quad \tau_3.\hat{\mathbf{h}} \in \{h, \top\} \\ \wedge \quad \tau_4.\hat{\mathbf{h}} \in \{h, \top\} \end{array} \right] \Rightarrow \left[\begin{array}{l} \tau_1 = \tau_2 \\ \wedge \quad \tau_1 < \top \\ \wedge \quad \forall w \in \mathbb{W} : (\tau_1.\mathbf{\Pi}(w) = 1 \Rightarrow \\ \tau_3.\mathbf{\Pi}(w) = \tau_4.\mathbf{\Pi}(w) = 1) \end{array} \right] \right)
\end{aligned}$$

Figure 3.9: Disjoint reachability property and disjoint reachability analysis.

all the v_2 objects would be retained in, for instance, a list). With statement (A), the type system can prove:

$$\emptyset, \langle 1, 1 \rangle, [v_1 \mapsto \tau_1, v_2 \mapsto \tau_2] \vdash \text{while}^1 \dots : [v_1 \mapsto \tau_1, v_2 \mapsto \tau_2], \{\tau_1 \supseteq \tau_2\}$$

where $\tau_1 = \langle h_1, \langle 1, 0 \rangle \rangle$ and $\tau_2 = \langle h_2, \langle 1, 1 \rangle \rangle$.

Statement (B) abstracts another common pattern where many objects allocated in the inner loop point to a single object allocated in the outer loop (for instance, parent or root pointers in tree data structures). Using statement (B), the type system can prove:

$$\emptyset, \langle 1, 1 \rangle, [v_1 \mapsto \tau_1, v_2 \mapsto \tau_2] \vdash \text{while}^1 \dots : [v_1 \mapsto \tau_1, v_2 \mapsto \tau_2], \{\tau_2 \supseteq \tau_1\}$$

where $\tau_1 = \langle h_1, \langle 1, 0 \rangle \rangle$ and $\tau_2 = \langle h_2, \langle 1, 1 \rangle \rangle$.

3.5 Disjoint Reachability Analysis

In this section, we present *disjoint reachability*, a form of object reachability that is used to compute conditional must not aliasing facts. We first formalize the *disjoint reachability property* in terms of the set of concrete heap effects C of a program execution and then present *disjoint reachability analysis* in terms of the set of abstract heap effects K of a well-typed program. Finally, we prove the disjoint reachability analysis sound with respect to the disjoint reachability property.

Consider the set of concrete heap effects C of a program execution. Recall from Section 3.3.2 that it contains an effect $(o_1 \triangleright o_2)$ if and only if some field of object o_1 was assigned object o_2 during the execution. We define the (non-reflexive) transitive closure of C , denoted C^+ , as follows:

Definition 3.7. (Closure of C) $C^+ = \bigcup_{n \geq 1} C^n$, where C^n is:

1. $C^1 = C$
2. If $(o_1 \triangleright o_2) \in C^n$ and $(o_2 \triangleright o_3) \in C$ then $(o_1 \triangleright o_3) \in C^{n+1}$

□

If $(o_1 \triangleright o_2) \in C^n$, then o_2 may be reachable from o_1 by n field dereferences. Hence, if $(o_1 \triangleright o_2) \in C^+$, then o_2 may be reachable from o_1 by one or more field dereferences. The disjoint reachability property is presented as the first equation in Figure 3.9. It states that $h \in DR_C(H)$ if and only if whenever an object o allocated at site h may be reachable by one or more field dereferences from each of objects o_1 and o_2 allocated at any sites in H , then o_1 and o_2 are one and the same object. In other words, the same object o allocated at site h is not reachable by one or more field dereferences from different objects o_1 and o_2 allocated at any sites in H .

We next define two notations we use in formulating our disjoint reachability analysis. We say types τ_1 and τ_2 are *compatible*, denoted $\tau_1 \sim \tau_2$, if they agree in all components where neither is $\top$:

Definition 3.8. $\tau_1 \sim \tau_2 \Leftrightarrow ((\tau_1.\hat{\mathbf{h}} = \tau_2.\hat{\mathbf{h}} \vee \tau_1.\hat{\mathbf{h}} = \top \vee \tau_2.\hat{\mathbf{h}} = \top) \wedge$
 $\forall w \in \mathbb{W} : (\tau_1.\mathbf{\Pi}(w) = \tau_2.\mathbf{\Pi}(w) \vee \tau_1.\mathbf{\Pi}(w) = \top \vee \tau_2.\mathbf{\Pi}(w) = \top))$

□

We say a type τ is *less than* $\top$ if no component of τ is $\top$:

Definition 3.9. $\tau < \top \Leftrightarrow (\tau.\hat{\mathbf{h}} \neq \top \wedge \forall w \in \mathbb{W} : \tau.\mathbf{\Pi}(w) \neq \top)$ $\square$

To define disjoint reachability analysis, we define the transitive closure of a set of abstract heap effects K , denoted K^+ , in a manner analogous to the transitive closure C^+ of a set of concrete heap effects C . Since abstract heap effects in K may correspond to multiple concrete heap effects, however, the transitive closure of K is more involved than that of C . Consider two abstract heap effects $(\tau_1 \supseteq \tau_2)$ and $(\tau_3 \supseteq \tau_4)$. If $\tau_2 \sim \tau_3$ then τ_2 and τ_3 may abstract the same object and hence there is a some transitive relationship between τ_1 and τ_4 . Simple transitivity is sound in all but one case: if $\tau_1.\mathbf{\Pi}(w) = \tau_4.\mathbf{\Pi}(w) = 1$ and either $\tau_2.\mathbf{\Pi}(w) \neq 1$ or $\tau_3.\mathbf{\Pi}(w) \neq 1$, then we cannot conclude that the objects abstracted by τ_1 and τ_4 are allocated in the same iteration of loop w . In this case it is sound to replace $\tau_4.\mathbf{\Pi}(w)$ by $\top$, ensuring that there is no information about the relative allocation times with respect to loop w .

Definition 3.10. (Closure of K) $K^+ = \bigcup_{n \geq 1} K^n$, where K^n is:

1. $K^1 = K$
2. If $(\tau_1 \supseteq \tau_2) \in K^n$ and $(\tau_3 \supseteq \tau_4) \in K$ and $\tau_2 \sim \tau_3$ then $(\tau_1 \supseteq \tau_5) \in K^{n+1}$ where

(a) $\tau_5.\hat{\mathbf{h}} = \tau_4.\hat{\mathbf{h}}$

(b) $\forall w \in \mathbb{W} :$

$$\tau_5.\mathbf{\Pi}(w) = \begin{cases} \top & \text{if } \tau_1.\mathbf{\Pi}(w) = 1 \wedge \tau_4.\mathbf{\Pi}(w) = 1 \wedge \\ & (\tau_2.\mathbf{\Pi}(w) \neq 1 \vee \tau_3.\mathbf{\Pi}(w) \neq 1) \\ \tau_4.\mathbf{\Pi}(w) & \text{otherwise} \end{cases}$$

$\square$

The following lemma proves the soundness of K^+ with respect to C^+ .

Lemma 3.11. *If $C \preceq K$ then $\forall n \geq 1 : C^n \preceq K^n$.*

Proof. By induction on n . See the proof of Lemma A.3 in Appendix A for details. $\square$

Our disjoint reachability analysis is presented as the second equation in Figure 3.9. Recall that Theorem 3.5 states that for any well-typed program with set of abstract heap effects K and for any execution of that program with set of concrete heap effects C , we have $C \preceq K$. The disjoint reachability analysis fact $h \in DR_K(H)$ is sufficient to prove the disjoint reachability property fact $h \in DR_C(H)$ for every $C \preceq K$.

The disjoint reachability analysis test is analogous to the disjoint reachability property test. It considers every relevant pair of abstract heap effects $(\tau_1 \supseteq \tau_3)$ and $(\tau_2 \supseteq \tau_4)$ in K^+ . Types τ_1 and τ_2 abstract objects o_1 and o_2 , respectively, in the disjoint reachability property test. Since o_1 and o_2 are objects allocated at sites in H , τ_1 and τ_2 must also abstract objects allocated at sites in H , that is, we must have $\tau_1.\hat{h}, \tau_2.\hat{h} \in H \cup \{\top\}$ (a type whose object allocation site is $\top$ may abstract an object allocated at any site, including one in H). Types τ_3 and τ_4 abstract the same object o in the disjoint reachability property test. Hence, we must have $\tau_3 \sim \tau_4$ (if this condition does not hold then types τ_3 and τ_4 can never both abstract the same object) and since o is allocated at site h , we must have $\tau_3.\hat{h}, \tau_4.\hat{h} \in \{h, \top\}$.

Intuitively, the test must now check that under the assumptions that τ_3 and τ_4 abstract the same object o and τ_1 and τ_2 abstract objects o_1 and o_2 such that o may be reachable by one or more field dereferences from each of o_1 and o_2 , it is the case that $o_1 = o_2$. This is done by the three conditions on the right-hand side of the implication. In order for τ_1 and τ_2 to abstract the same object, the types must first be equal ($\tau_1 = \tau_2$), and secondly, they must have no top elements ($\tau_1 < \top$), since a top element in any position of the loop vector or in the object allocation site would allow the type to abstract more than one object and hence the objects abstracted by the two types could not be shown to be the same. The third condition is an implication whose antecedent $\tau_1.\Pi(w) = 1$ (and hence $\tau_2.\Pi(w) = 1$ since we have just ensured $\tau_1 = \tau_2$) states that τ_1 and τ_2 may still abstract different objects o_1 and o_2 , namely, o_1 and o_2 may be allocated in different iterations of loop w , but in this case, the consequent $\tau_3.\Pi(w) = \tau_4.\Pi(w) = 1$ ensures that $o_1 = o_2$, by the following reasoning. Since $(\tau_1 \supseteq \tau_3)$ in K^+ and $\tau_1.\Pi(w) = \tau_3.\Pi(w) = 1$, the object o abstracted by τ_3 and the object o_1 abstracted by τ_1 are guaranteed to be allocated in the same iteration of loop w (recall that the computation of K^+ enforces this condition). Likewise, since

$(\tau_2 \supseteq \tau_4)$ in K^+ and $\tau_2.\mathbf{\Pi}(w) = \tau_4.\mathbf{\Pi}(w) = 1$, the object o abstracted by τ_4 and the object o_2 abstracted by τ_2 are guaranteed to be allocated in the same iteration of loop w . Combining the above two arguments, it follows that o_1 and o_2 are allocated in the same iteration of loop w , whence $o_1 = o_2$.

We are now ready to prove the soundness of our disjoint reachability analysis with respect to the disjoint reachability property. The key lemma in this proof is the following:

Lemma 3.12. *If $C \preceq K$ and $h \in DR_K(H)$ then $h \in DR_C(H)$.*

Proof. From Lemma 3.11 and the definitions of DR_C and DR_K in Figure 3.9. See the proof of Lemma A.4 in Appendix A for details. $\square$

Theorem 3.13. (Soundness of Disjoint Reachability Analysis)

If $s, \emptyset, \lambda w.0, \lambda v.\perp, \square \Downarrow \pi, \rho, \sigma, C$ and $\emptyset, \mathbf{\Pi}, \Gamma \vdash s : \Gamma', K$ and $h \in DR_K(H)$ then $h \in DR_C(H)$.

Proof. Immediate from Theorem 3.5 and Lemma 3.12. $\square$

Recall from the end of Section 3.4 that our type and effect system derives the set of abstract heap effects $K = \{\langle h_1, \langle 1 \rangle \rangle \supseteq \langle h_2, \langle 1 \rangle \rangle\}$ for Example 3.1. In this simple example, we have $K^+ = K$, and using the disjoint reachability analysis in Figure 3.9, we have $h_2 \in DR_K(\{h_1\})$, that is, the same object allocated at site h_2 is not reachable by one or more field dereferences from different objects allocated at site h_1 .

Now consider the nested loops in Example 3.6. Using statement (A), we have $K = \{\langle h_1, \langle 1, 0 \rangle \rangle \supseteq \langle h_2, \langle 1, 1 \rangle \rangle\}$ and again $K^+ = K$. Because the loop vector of the type on the left side of this effect does not have a $\top$ in any position and because the loop vector of the type on the right side of this effect has a 1 in every position where that on the left side has a 1, our analysis infers that the same object allocated at site h_2 in the inner loop is not reachable by one or more field dereferences from different objects allocated at site h_1 in the outer loop; thus $h_2 \in DR_K(\{h_1\})$. Finally, using statement (B), we have $K = \{\langle h_2, \langle 1, 1 \rangle \rangle \supseteq \langle h_1, \langle 1, 0 \rangle \rangle\}$ and $K^+ = K$. Because the loop vector of the type on the right side of this effect has a 0 in a position where that on the left side has a 1, our analysis infers that the same object allocated at site h_1

in the outer loop may be reachable by one or more field dereferences from different objects allocated at site h_2 in the inner loop; thus $h_1 \notin DR_K(\{h_2\})$.

3.6 Practical Considerations

Our presentation thus far has elided some features that are indispensable for precision in a static race detection algorithm for a realistic language like Java. We discuss these features in this section.

As discussed in Chapter 2, our static race detection algorithm uses a k -object-sensitive analysis [60, 61] for precision. Hence, object allocation sites in our presentation become points-to sets computed by our k -object-sensitive analysis, namely, sets of sequences of upto k object allocation sites $[h_n :: \dots :: h_1]$ where $1 \leq n \leq k$. Also, the join operator in our presentation, which yields $\top$ when applied to two different object allocation sites, is now applied to points-to sets and computes set union.

Our WHILE language lacks methods. For Java programs, we must handle not only loops but also methods since the effect of looping may be achieved by recursive methods. Our solution is to transform all loops to tail-recursive methods upfront. Our k -object-sensitive analysis constructs a context sensitive call graph, that is, each method may be called in multiple abstract contexts. Then, loop vectors in our presentation become vectors containing one position for each (m, c) pair such that method m may be called in abstract context c . This enables us to handle both loops and recursive methods uniformly.

Combining the features discussed so far, a type $\langle \hat{h}, \Pi \rangle$ which consisted of an object allocation site and a loop vector now consists of a points-to set and a vector of (method, abstract context) pairs.

Finally, another source of imprecision in our presentation is the treatment of heap reads; consider the following example:

1. `v1.f = v2`
2. `... no writes to aliases of v1.f ...`
3. `v3 = v1.f`

According to Rule (3.14) in Figure 3.8, the type of $\mathbf{v3}$ on line 3 is $\langle \lambda w. \top, \top \rangle$, that is, no useful information is known for $\mathbf{v3}$. Unfortunately, this rule is too coarse in practice, as there are situations similar to the one given above in real-world Java programs.

To improve precision, we compute flow sensitive, must alias facts, for instance, after line 3, we want to know that $\mathbf{v3} = \mathbf{v2}$, that is, the type of $\mathbf{v3}$ has the same method vector and points-to set as that of $\mathbf{v2}$. The approach we use is a standard (but interprocedural) dataflow algorithm to track must alias facts on names of the form $\mathbf{v.f1.f2.f3...}$; there are similar algorithms in the literature [24].

This extension also introduces a new problem for a race detection algorithm that aims to be sound. Consider the read again on line 3 above. The conclusion that $\mathbf{v3} = \mathbf{v2}$ after line 3 is only valid if line 2 contains no writes to aliases of $\mathbf{v1.f}$ and, moreover, no other thread writes to an alias of $\mathbf{v1.f}$. It is not surprising that a flow sensitive computation must reason about potential races, but it does lead to a recursively defined notion of race detection, as computing the set of races now depends on knowing the set of races to begin with. We solve this problem by using our thread escape analysis. In particular, if the analysis determines that $\mathbf{v1}$ always points to a thread-local object, then we can conclude that $\mathbf{v3} = \mathbf{v2}$ after line 3 (presuming line 2 contains no writes to aliases of $\mathbf{v1.f}$). If the analysis determines that $\mathbf{v1}$ may point to a thread escaping object, however, then we kill the must alias dataflow fact $\mathbf{v3} = \mathbf{v2}$, that is, the type of $\mathbf{v3}$ after line 3 is $\langle O, \lambda w. \top \rangle$ where O is the points-to set of $\mathbf{v3}$ computed by our thread escape analysis.

3.7 Conditional Must Not Alias Analysis

We now present our conditional must not alias analysis in the framework of our static race detection algorithm presented in Chapter 2. The algorithm begins with a relation `originalRaces` which is an initial over-approximation of the set of races in the given Java program. It contains each tuple of the form $(c_1^t, c_1^m, e_1, c_2^t, c_2^m, e_2)$ such that abstract threads c_1^t and c_2^t may execute statements e_1 and e_2 in abstract contexts c_1^m and c_2^m of their containing methods, respectively. The abstract threads

and abstract contexts arise from k -object-sensitive analysis [60, 61], namely, each of them is either a distinguished element ϵ (which may be viewed both as the thread abstracting the implicit main thread of the program and as the lone abstract context of the main method) or a sequence of at most k object allocation sites $[h_n :: \dots :: h_1]$ ($1 \leq n \leq k$). Statements e_1 and e_2 in each tuple either access the same instance or static field of a class or they both access array elements. In either case, at least one of those accesses is a write. These are the only cases in which a pair of statements in a Java program may be involved in a race. The set of real races is typically a tiny fraction of `originalRaces`. Our original race detection algorithm uses the following four static analyses which check different conditions for race freedom to eliminate tuples from `originalRaces`:

- A may alias analysis eliminates each tuple such that e_1 and e_2 never access the same memory location in abstract contexts c_1^m and c_2^m , respectively. The set of tuples retained is denoted `aliasingRaces`.
- A thread escape analysis eliminates each tuple such that either e_1 always accesses a thread local memory location in abstract context c_1^m , or e_2 always accesses a thread local memory location in abstract context c_2^m , or both. The set of tuples retained is denoted `escapingRaces`.
- A may-happen-in-parallel analysis eliminates each tuple such that the thread structure of the program prevents abstract threads c_1^t and c_2^t from simultaneously executing e_1 and e_2 in abstract contexts c_1^m and c_2^m , respectively. The set of tuples retained is denoted `parallelRaces`.
- A lockset analysis eliminates each tuple such that either c_1^t and c_2^t *may* abstract one and the same thread or a common lock *may* be held by the threads while executing e_1 and e_2 in abstract contexts c_1^m and c_2^m , respectively. The set of tuples retained is denoted `unlockedRaces`.

Finally, our algorithm computes the set of potential races to be reported, denoted `ultimateRaces`, as containing each tuple in `originalRaces` that could not be eliminated by any of the above four analyses:

$$\text{ultimateRaces} = \text{aliasingRaces} \cap \text{escapingRaces} \cap \\ \text{parallelRaces} \cap \text{unlockedRaces}$$

The algorithm is precise in practice in that `ultimateRaces` contains few false positives, but it is unsound in that it may have false negatives because, as discussed in Section 3.1, the lockset analysis it uses is unsound, namely, tuples not retained in `unlockedRaces` are not necessarily race-free. We next describe how to compute `unlockedRaces` soundly using conditional must not alias analysis, rendering our race detection algorithm sound.

Our conditional must not alias analysis consists of three independent analyses each of which targets a different locking idiom to prove tuples in `originalRaces` race-free. All three analyses are based on the concept of conditional must not aliasing but differ operationally. We denote the sets of tuples retained by these three analyses as `globalUnlockedRaces`, `localUnlockedRaces`, and `threadUnlockedRaces`. Then, `unlockedRaces` is computed soundly as follows:

$$\text{unlockedRaces} = \text{globalUnlockedRaces} \cap \text{localUnlockedRaces} \cap \\ \text{threadUnlockedRaces}$$

We illustrate these three analyses using the example Java program from Figure 3.1. As discussed in Section 3.2, the example illustrates four scenarios in all: three scenarios in which expression `thrd` is `T.g[*]`, that is, each child thread non-deterministically chooses a `T` object corresponding to any of the child threads, and expression `lock` is one of `T.g`, `vt`, and `ve`, illustrating the coarse-, fine-, and medium-grained locking styles, respectively, plus a fourth scenario in which expression `thrd` is `this`, that is, each child thread deterministically chooses the `T` object corresponding to itself. In each of these scenarios, relation `originalRaces` includes the tuple $([h2], [h2], e, [h2], [h2], e)$ since each child thread, abstracted by `[h2]`, executes statement `e`

in abstract context [h2]. None of the other sound analyses in our race detection algorithm, namely, the may alias analysis, the thread escape analysis, and the may-happen-in-parallel analysis, can eliminate this tuple. We next present the three analyses comprising our conditional must not alias analysis and show how different of those analyses eliminate this tuple under the different scenarios discussed above.

3.7.1 Computation of globalUnlockedRaces

This analysis specializes in proving race freedom in the presence of global, uniquely named locks. An example is the coarse-grained locking scenario in the above example. This style of locking is fairly common in real-world Java programs. Java programmers not only create global locks explicitly but also use such locks created by the virtual machine, for instance, by using `static synchronized` methods or by synchronizing on `class` fields.

This analysis is similar to the lockset analysis in our original race detection algorithm from Chapter 2. Conceptually, for each $(c_1^t, c_1^e, e_1, c_2^t, c_2^e, e_2) \in \text{originalRaces}$, the analysis considers each pair of cycle-free paths in the context-sensitive call graph of the program of the form:

$$\begin{aligned} (c_1^t, m_1^t) &\rightarrow^{i_1} \dots \rightarrow^{i_j} (c_1^e, m_1^e) \\ (c_2^t, m_2^t) &\rightarrow^{i'_1} \dots \rightarrow^{i'_k} (c_2^e, m_2^e) \end{aligned}$$

where edge $(c, m) \rightarrow^i (c', m')$ means that call site i in abstract context c of its containing method m may call method m' in abstract context c' , and m_1^t and m_2^t are the root methods of abstract threads c_1^t and c_2^t , respectively, and m_1^e and m_2^e are the methods containing e_1 and e_2 , respectively. Suppose a lock is held along each path by a synchronized statement lexically enclosing one of $i_1, \dots, i_j$ or e_1 (resp. one of $i'_1, \dots, i'_k$ or e_2) on local variable v_1^l (resp. v_2^l) in abstract context c_1^l (resp. c_2^l). Let $P(v, c)$ denote the points-to set of local variable v in abstract context c of its containing method. Then, this analysis eliminates the above tuple provided the following condition holds:

- $P(v_1^l, c_1^l) = P(v_2^l, c_2^l) = \{o\}$ and o abstracts at most one concrete object in any

execution.

Note that this lockset analysis, unlike that in our original race detection algorithm, is sound because it only considers lock expressions whose points-to sets are singleton and also checks that the lone abstract object in each such points-to set abstracts at most one concrete object in any execution. The set of tuples retained by this analysis is denoted `globalUnlockedRaces`.

Consider the coarse-grained locking scenario in the above example, that is, suppose expression `lock` is `T.g` (and expression `thrd` is `T.g[*]`). Consider the tuple $([h2], [h2], e, [h2], [h2], e)$ in `originalRaces`. There is only one pair of paths to consider for this tuple and both paths in this pair are of the form $([h2], \text{start}) \rightarrow ([h2], \text{run})$, that is, each child thread begins executing in abstract context `[h2]` of the `start` method of class `java.lang.Thread` which calls the method containing statement `e`, that is, the `run` method of class `T`, in the same abstract context. A lock is held along this path by the synchronized statement lexically enclosing `e` on local variable `v1` in abstract context `[h2]`. We have $P(v1, [h2]) = \{[h1]\}$, and moreover, site `h1` which allocates the array of `T` objects is executed exactly once in every execution. Hence, the above condition holds and the tuple is eliminated by this analysis.

3.7.2 Computation of `localUnlockedRaces`

This analysis specializes in proving race freedom in the presence of non-global locks. Examples include the fine-grained and medium-grained locking scenarios in the above example. This style of locking is also fairly common in real-world Java programs, for example, a lock on an object `o` is often used to guard fields of `o` (the fine-grained locking scenario) or to guard fields of objects pointed to by fields of `o` and so on (the medium-grained locking scenario).

Conceptually, for each $(c_1^t, c_1^e, e_1, c_2^t, c_2^e, e_2) \in \text{originalRaces}$ such that e_1 and e_2 are of the form $v_1^e.f$ and $v_2^e.f$ where v_1^e and v_2^e are local variables and f is an instance field or a hypothetical field that is regarded as accessed whenever an array element is accessed, the analysis considers each pair of cycle-free paths in the context-sensitive

call graph of the program of the form:

$$\begin{aligned} (c_1^t, m_1^t) &\rightarrow^{i_1} \dots \rightarrow^{i_j} (c_1^e, m_1^e) \\ (c_2^t, m_2^t) &\rightarrow^{i'_1} \dots \rightarrow^{i'_k} (c_2^e, m_2^e) \end{aligned}$$

where edge $(c, m) \rightarrow^i (c', m')$ means that call site i in abstract context c of its containing method m may call method m' in abstract context c' , and m_1^t and m_2^t are the root methods of abstract threads c_1^t and c_2^t , respectively, and m_1^e and m_2^e are the methods containing e_1 and e_2 , respectively. Suppose a lock is held along each path by a synchronized statement lexically enclosing one of $i_1, \dots, i_j$ or e_1 (resp. one of $i'_1, \dots, i'_k$ or e_2) on local variable v_1^l (resp. v_2^l) in abstract context c_1^l (resp. c_2^l).

We first consider the easier case of fine-grained locking. This analysis eliminates the above tuple provided the following condition holds:

- v_1^e is must aliased with v_1^l and, similarly, v_2^e is must aliased with v_2^l .

Recall from Section 3.6 that we already compute must alias facts of this form. The reason the above tuple is race-free if the above condition holds is as follows. The argument presumes that there is a race under the above condition and derives a contradiction. In order for a race to occur, two different threads must execute the above paths and acquire locks on *different* objects denoted by v_1^l and v_2^l (otherwise there is no race). The threads must proceed to execute the paths until they reach e_1 and e_2 , respectively, where they must access field f of the *same* object denoted by v_1^e and v_2^e (otherwise there is no race). Now, under the assumption that the lock objects are different and the above condition that v_1^e (resp. v_2^e) is must aliased with v_1^l (resp. v_2^l), it follows that the accessed objects are different, a contradiction.

Consider the fine-grained locking scenario in the above example, that is, suppose expression `lock` is `ve` (and expression `thrd` is `T.g[*]`). Consider the tuple $([h2], [h2], e, [h2], [h2], e)$ in `originalRaces`. As discussed above, there is only one pair of identical paths to consider, and along this path, each thread holds a lock on the object denoted by local variable `v1` while accessing the object denoted by local variable `ve`. In this case, we have `v1 = ve`. Hence, the above condition holds and the tuple is eliminated by this analysis.

We next consider the harder case of medium-grained locking. Once again consider the above tuple. This analysis eliminates it provided the following conditions hold:

- v_1^e is obtained by one or more field dereferences from v_1^l , and similarly for v_2^e and v_2^l .
- $(P(v_1^e, c_1^e) \cap P(v_2^e, c_2^e)) \subseteq DR_K(P(v_1^l, c_1^l) \cup P(v_2^l, c_2^l))$ where K is the set of abstract heap effects of the program.

In the first condition, we mean that v_1^e is assigned $v_1^l.f_1$ or $v_1^l.f_1.f_2$ or $v_1^l.f_1.f_2.f_3$ and so on (and similarly for v_2^e and v_2^l). Such expressions are split using temporary local variables to store intermediate results and once again, we use must alias facts as illustrated in Section 3.6 to check whether this condition holds. Note that the second condition uses our disjoint reachability analysis from Section 3.5. The reason the above tuple is race-free if the above conditions hold is as follows. As in the case for fine-grained locking above, the argument presumes that there is a race under the above conditions and derives a contradiction. In order for a race to occur, two threads must execute the above paths and acquire locks on *different* objects denoted by v_1^l and v_2^l (otherwise there is no race). Since the points-to information we compute is sound, these objects must be abstracted by abstract objects in the set $L = (P(v_1^l, c_1^l) \cup P(v_2^l, c_2^l))$. The threads must proceed to execute the paths until they reach e_1 and e_2 , respectively, where they must access field f of the *same* object denoted by v_1^e and v_2^e (otherwise there is no race). Since the points-to information we compute is sound, this object must be abstracted by some abstract object in the set $E = (P(v_1^e, c_1^e) \cap P(v_2^e, c_2^e))$. Now, from the first condition above, it follows that the same accessed object is obtained by one or more field dereferences from each of the two different lock objects. But from the second condition above ($E \subseteq DR_K(L)$), it follows that the same accessed object is unreachable by one or more field dereferences from each of the two different lock objects, a contradiction.

Consider the medium-grained locking scenario in the above example, that is, suppose expression `lock` is `vt` (and expression `thrd` is `T.g[*]`). Consider the tuple $([h2], [h2], e, [h2], [h2], e)$ in `originalRaces`. As discussed above, there is only one pair of identical paths to consider, and along this path, each thread holds a lock

on the object denoted by local variable `v1` while accessing the object denoted by local variable `ve`. In this case, we have `v1 = vt` and `ve = vt.f1`, satisfying the first condition above. Also, we have $P(v1, [h2]) = \{[h2]\}$, $P(ve, [h2]) = \{[h3]\}$, and it is easy to see that $\{[h3]\} \subseteq DR_K(\{[h2]\})$, satisfying the second condition above. Thus, both conditions are satisfied and hence the tuple is eliminated by this analysis.

In summary, this analysis handles both the fine-grained and medium-grained locking cases, using the disjoint reachability analysis only in the latter case. The set of tuples collectively retained by this stage is denoted `localUnlockedRaces`.

3.7.3 Computation of `threadUnlockedRaces`

This analysis is identical to the analysis presented in Section 3.7.2 except that it reasons about thread objects instead of lock objects, in particular, it tries to prove that whenever two thread objects are different, the objects they access are also different. We illustrate this analysis by the final scenario in the above example, in which expression `thrd` is `this`. (Recall that locking is redundant in this scenario and so the value of expression `lock` is immaterial.) Consider the tuple $([h2], [h2], e, [h2], [h2], e)$ in `originalRaces`. As discussed above, there is only one pair of identical paths to consider, and along this path, each thread corresponding to the object denoted by local variable `vt` accesses the object denoted by local variable `ve`. In this case, we have `ve = vt.f1`, satisfying the first condition. Also, $P(vt, [h2]) = \{[h2]\}$, $P(ve, [h2]) = \{[h3]\}$, and $\{[h3]\} \subseteq DR_K(\{[h2]\})$, satisfying the second condition. Thus, both conditions are satisfied and hence the tuple is eliminated by this analysis.

3.8 Experiments

We have evaluated our conditional must not alias analysis on a suite of seven multi-threaded Java programs. Table 3.1 provides a brief description of each program along with the number of classes and the number of bytecodes in the program. The numbers correspond to code that is deemed reachable from the main method of each program in a context insensitive call graph that is computed by Spark [52], a 0-CFA-based may

	# classes	# bytecodes	brief description
<code>elevator</code>	1066	109831	Discrete event simulator
<code>tsp</code>	1068	110582	TSP solver from ETH
<code>hedc</code>	1592	278010	Web crawler from ETH
<code>ftp</code>	1905	348813	Apache FTP server
<code>pool</code>	1121	122187	Apache pooling library
<code>jdbf</code>	1739	291392	Object-relational mapping system
<code>jtds</code>	1801	301231	JDBC driver

Table 3.1: Benchmarks.

alias analysis with on-the-fly call graph construction provided in the Soot compiler framework [81]. The experiments were performed on a 2.4GHz machine with 4GB memory.

Figure 3.2 presents the total running time of our previous race detection algorithm using the unsound lockset analysis under the “unsound” column and the total running time of our current race detection algorithm using conditional must not alias analysis under the “sound” column. Recall from Chapter 2 that our implementation of k -object-sensitive analysis is parameterized by a positive integer k that may be instantiated differently for different programs; we have used $k = 3$ for each program. We call this variant of k -object-sensitive analysis *non-adaptive* since it uses the same k value for all object allocation sites in the program. Our current race detection algorithm runs out of memory using $k = 3$ for our single largest benchmark, Apache Derby (`derby`), which our previous race detection algorithm is capable of analyzing. In Chapter 4, we present an *adaptive* variant of k -object-sensitive analysis that allows different k values to be used for different object allocation sites in the same program, which enables a demand-driven race detection algorithm that is capable of analyzing `derby`. The demand-driven algorithm strikes a good trade-off between scalability and precision by using bigger k values for a few sites and smaller k values for the vast majority of sites in the program.

Table 3.3 compares the effectiveness of conditional must not alias analysis to the lockset analysis used in our previous race detection algorithm. The “unsound” column presents the fraction of original race pairs that were deemed race-free by the

	unsound	sound
<code>elevator</code>	4m57s	12m14s
<code>tsp</code>	4m12s	10m41s
<code>hedc</code>	13m11s	38m12s
<code>ftp</code>	17m32s	54m15s
<code>pool</code>	4m17s	9m43s
<code>jdbf</code>	9m41s	29m31s
<code>jtds</code>	12m52s	37m12s

Table 3.2: Experimental results: Comparison of running time.

lockset analysis while the “total” sub-column under the “sound” column presents the fraction of original race pairs that were proven race-free by conditional must not alias analysis. Note that the latter numbers are smaller than the former because (1) the lockset analysis is unsound (and therefore can eliminate pairs that are in fact not race-free) and (2) the conditional must not alias analysis is sound (and therefore can fail to eliminate pairs that are in fact race-free). Each pair proven race-free by conditional must not alias analysis is further categorized under the “global”, “local”, and “thread” sub-columns, depending upon which of the three sub-analyses computing `globalUnlockedRaces`, `localUnlockedRaces`, and `threadUnlockedRaces`, respectively, eliminated the pair. The sum of the three columns typically exceeds 100% because certain pairs are proven race-free by more than one of the above three sub-analyses. For instance, library classes like `java.util.Vector` and `java.io.*` use synchronization extensively to ensure thread-safety when used in multi-threaded contexts but application classes often use them in single-threaded contexts. Such code causes pairs to appear under both the “local” and “thread” sub-columns.

Finally, Table 3.4 presents the impact of conditional must not alias analysis on the ultimate race pairs reported by our race detection algorithm. In particular, the “unsound” column presents the number of ultimate race pairs reported by our previous race detection algorithm using the lockset analysis while the “sound” column presents the number of ultimate race pairs reported by our current race detection algorithm using conditional must not alias analysis. Each column is further partitioned into “real” and “false”, denoting the number of real races and false positives, respectively.

	unsound	sound			
		global	local	thread	total
elevator	39%	42%	35%	34%	37%
tsp	41%	43%	34%	32%	39%
hedc	43%	39%	39%	29%	40%
ftp	48%	36%	42%	32%	45%
pool	45%	31%	44%	26%	41%
jdbf	40%	32%	46%	31%	36%
jtds	47%	34%	45%	29%	42%

Table 3.3: Experimental results: Comparison of numbers of unlocked race pairs.

	unsound		sound	
	real	false	real	false
elevator	0	0	0	1
tsp	6	2	11	8
hedc	124	19	159	56
ftp	213	34	271	107
pool	27	0	29	2
jdbf	258	8	282	20
jtds	241	7	288	34

Table 3.4: Experimental results: Comparison of numbers of ultimate race pairs.

The increase in the number of real races reported by our current algorithm over that reported by our previous algorithm is because of false negatives resulting from the unsound lockset analysis used by our previous algorithm. Likewise, the increase in the number of false positives is because of false positives resulting from the sound conditional must not alias analysis used by our current algorithm.

3.9 Related Work

We have already discussed related work on race detection in Chapter 2. In this section, we survey the literature on problems related to conditional must not aliasing and disjoint reachability. We begin by noting that our notion of loop vectors harkens back to the ideas of *iteration space* and *dependence distance* in work on vectorizing

compilers. Loop vectors are points in the iteration space of the program and our algorithm can be thought of as tracking dependent statements of distance 0 (that is, in the same iteration). We are not aware of any deeper connections to the large literature on program parallelization; our focus is on pointer-based data structures while parallelizing compilers focus primarily on array-based data structures.

Disjoint reachability for pointer-based data structures (e.g., proving that two lists constructed of separate elements are disjoint) is an old problem. Algorithms in this area range from flow sensitive approximations of heap shape ([21] is an early example) to very powerful decision procedures [50, 58]. Our notion of disjoint reachability is less precise (e.g., it is flow insensitive) but easier to scale to large programs and gives good results for static race detection for Java programs.

Ownership types express the idea that among all pointers to an object, one is often special in that it has more operations than other pointers. In the context of static race detection, ownership can be used to prove encapsulation (that an object is the exclusive access to another object), which in turn can prove conditional must not aliasing facts: if two objects are distinct, then any objects they encapsulate must be distinct. There are algorithms for inferring ownership [43] and encapsulation [38] and ownership types have been exploited in static race detectors [11]. While encapsulation is sufficient to prove conditional must not aliasing, it is not necessary. Roughly speaking, ownership/encapsulation are properties of how an object is constructed, while conditional must not aliasing also considers the specific pointers through which an object is used; this more refined treatment can fully automatically check race freedom for objects that are not encapsulated and without ownership types. Indeed, our approach is fully automatic, while to the best of our knowledge current ownership systems for static race detection require at least some manual annotations.

Another related approach is that a *correlation* exists between two objects if they are used consistently together [69]. For race detection, correlation means a particular lock is always used to guard a particular memory location. *Correlation analysis* infers which locks always guard which memory locations. Our approach does not require locks and memory locations to be correlated; because each pair of statements potentially involved in a race is handled separately, different locks may be used to prove

race freedom for the same memory location accessed by different pairs of statements.

Chapter 4

A Demand-Driven Approach

This chapter presents a demand-driven approach to static race detection for Java. The key idea underlying the approach is the application of an adaptive k -object-sensitive may alias analysis capable of using different k values for different object allocation sites in the same program, in contrast to our previous approach which employs a non-adaptive k -object-sensitive may alias analysis that uses the same k value for all sites, requiring that approach to sacrifice scalability, precision, or soundness. We demonstrate the effectiveness of our demand-driven approach at striking a good balance between scalability and precision without sacrificing soundness on a suite of eight multithreaded Java programs.

4.1 Introduction

Race detection algorithms for shared-memory multithreaded programs using lock-based synchronization involve checking necessary conditions for a race such as whether different threads can access the same memory location without holding a common lock. Such algorithms must reason about aliasing relationships between threads, locks, and memory locations, as alluded to by requirements such as the threads must be different, any locks they hold must be different, and the memory location they access must be the same.

Much recent work on race detection has focused on Java which supports the style

of concurrency embodied in shared-memory multithreaded programs using lock-based synchronization [11, 12, 28, 29, 73]. Java, an object-oriented language, pervasively uses objects for this purpose. For instance, threads are themselves objects of (any subclass of) `java.lang.Thread`. Locks are even more pervasive since a lock can be held on *any* object. Finally, the vast majority of memory locations on which races can occur in a Java program are either array elements or instance fields of objects, and since arrays in Java are themselves objects, array elements may be viewed as instance fields of objects.¹

Since threads, locks, and memory locations are all accessed through objects, it is natural for a race detection algorithm for Java to reason about aliasing relationships between objects. We showed in Chapters 2 and 3 that a relatively recent form of may alias analysis for Java, called k -object-sensitive analysis introduced by Milanova et al. [60, 61] is effective at statically approximating such relationships. The variant of the analysis we employed, however, is *non-adaptive* in that it uses the same k value for all object allocation sites in a given program, which in turn requires our race detection approach to sacrifice scalability, precision, or soundness, as smaller k values hurt precision and larger k values impair scalability. In particular, we have shown how to achieve scalability and precision at the cost of soundness in Chapter 2 and how to achieve precision and soundness at the cost of scalability in Chapter 3, but we have not shown how to achieve a good trade-off between scalability and precision while retaining soundness.

In this chapter, we extend our static race detection algorithm to employ an *adaptive* k -object-sensitive analysis capable of using different k values for different object allocation sites in the same program. The algorithm is *demand-driven*: it guides the k -object-sensitive analysis to use bigger k values for sites where it deems higher precision is necessary and smaller k values for sites where it deems lower precision suffices. In practice, it uses bigger k values for few sites and smaller k values for the vast majority of sites, thereby striking a good trade-off between scalability and precision while retaining soundness.

¹The remaining memory locations on which races can occur in Java are static fields, which may be viewed as global variables.

Given the complexity of our static race detection algorithm which uses four different analyses, namely, may alias analysis, thread escape analysis, may-happen-in-parallel analysis, and conditional must not alias analysis, our demand-driven approach is not optimal—it may choose bigger k values for sites where smaller k values suffice and, more likely and worse, it may choose smaller k values for sites where bigger k values are necessary—but it is effective in practice. We have implemented and applied it to eight multithreaded Java programs and its key contributions may be summarized as follows:

1. It is scalable in practice: it uses a k value greater than one for less than 0.01% of all sites in each program. In contrast, our earlier algorithm using a k value of more than one for all sites runs out of memory for our largest program.
2. It is precise in practice: for five of the seven smaller programs, it reports equal or fewer false races than our earlier algorithm using a k value of three for all sites, and for the largest program, it reports five times fewer false races than our earlier algorithm using a k value of one for all sites.
3. It is fully automatic: it saves the user the effort of providing a k value to be used for all sites in each program, which is required in our earlier algorithm and involves a manual iterative process in which the user supplies increasingly bigger k values until it yields precise enough results or fails to scale.

The rest of this chapter is organized as follows. Section 4.2 illustrates our approach by means of an example Java program. Section 4.3 presents a language to formalize our approach. Section 4.4 presents our implementation of k -object-sensitive analysis. Section 4.5 presents our demand-driven race detection algorithm. Section 4.6 demonstrates its effectiveness on our benchmark suite. Finally, Section 4.7 discusses related work.

4.2 Example

In this section, we present a multithreaded Java program that we use as the running example to illustrate our approach. The program, shown in Figure 4.1, begins by

```

public class T extends java.lang.Thread {
    public static void main(String[] args) {
hg:      T.g = new C[*];
          for (int i = 0; i < *; i++)
              T.g[i] = C.newInstance();
          for (int i = 0; i < *; i++) {
ht:      T t = new T();
          t.start();
          }
    }
    private static C[] g;
    private C f1;
    public T() { this.f1 = C.newInstance(); }
    public void run() {
        C v1 = this.f1;
        int[] v2 = v1.f2;
p1:      v2[*] = *;
        C[] v3 = T.g;
        C v4 = v3[*];
        int[] v5 = v4.f2;
        synchronized (lock) {    // Choices for lock: v3, v4, v5
p2:      v5[*] = *;
        }
    }
}

public class C {
    public int[] f2;
    private C() {
hi:      this.f2 = new int[*];
    }
    public static C newInstance() {
hc:      return new C();
    }
}

```

Figure 4.1: Example multithreaded Java program.

executing the `main` method of class `T` in an implicit main thread. The main thread first creates an array of `C` objects and stores it in static field `g` of class `T`, which may be viewed as a global variable. The main thread then executes two loops. The first loop initializes the elements of the created array with fresh `C` objects while the second loop creates a bunch of `T` objects each of whose field `f1` is set to a fresh `C` object. Whenever a `C` object is created in each iteration of either loop, its field `f2` is set to a fresh integer array. The resulting data structure is shown in Figure 4.2. For convenience, we label each object in the figure with the site at which it was allocated. In each iteration of the second loop, the main thread also calls the `start` method of superclass `java.lang.Thread` on the `T` object created in the same iteration. Each call to the `start` method, whose body is not shown, spawns a fresh child thread which calls the `run` method of class `T` on the corresponding `T` object. The calls to the `start` method are asynchronous. Thus, all child threads execute the `run` method in parallel.

Each child thread first executes the statement labeled `p1`, which writes to a non-deterministically chosen element of the integer array pointed to by field `f2` of the `C` object pointed to by field `f1` of the `T` object corresponding to that thread. Observe that statement `p1` executed by any two different child threads writes to different memory locations and so there are no races.

Each child thread next non-deterministically chooses one of the `C` objects in the global array of `C` objects stored in static field `T.g`, acquires a lock on the object denoted by expression `lock`, left unspecified for now, executes the statement labeled `p2` which writes to a non-deterministically chosen element of the integer array pointed to by field `f2` of the chosen `C` object, and releases the lock. Statement `p2` executed by two different child threads may write to the same memory location, namely, both threads may choose the same `C` object in the global array and then choose the same element of the integer array. Thus, if a common lock is not held by the two threads, the program has a race. We consider three choices for the object denoted by expression `lock`: the global array of `C` objects, the non-deterministically chosen `C` object, and the integer array pointed to by field `f2` of the chosen `C` object. Observe that in each of these cases, each pair of child threads hold a common lock if they write to the same memory location, and so there are no races.

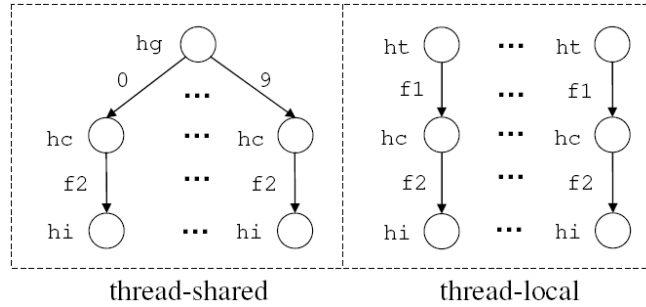


Figure 4.2: Data structure built by example program.

Our example program highlights object-oriented idioms used in real multithreaded Java programs that motivate our demand-driven race detection approach. First, such programs create multiple threads at run-time that operate concurrently on two kinds of deeply nested data structures:

- thread-local structures like the `T` objects and the identical but disjoint parts of the heap reachable from them, depicted on the right-hand side of Figure 4.2. Concurrent accesses such as those by statement `p1` from different threads to such structures that are local to those threads do not need any synchronization.
- thread-shared structures like the global array of `C` objects stored in static field `T.g` and the parts of the heap reachable from it, depicted on the left-hand side of Figure 4.2. Concurrent accesses such as those by statement `p2` from different threads to such structures use lock-based synchronization to prevent races.

Secondly, the threads heavily reuse the same library code to create and manipulate parts of these structures. For instance, class `C` may be a library class like `java.lang.Integer` or `java.util.ArrayList` that may be used to create and manipulate parts of both thread-local and thread-shared structures. Concurrent accesses such as those by statements `p1` and `p2` from different threads to thread-local and thread-shared structures, respectively, do not need any synchronization since they operate on disjoint structures, although parts of both are created using the same class `C`.

Such heavily reused code operating on deeply nested data structures poses significant challenges to static race detection, which we elucidate next by giving specific

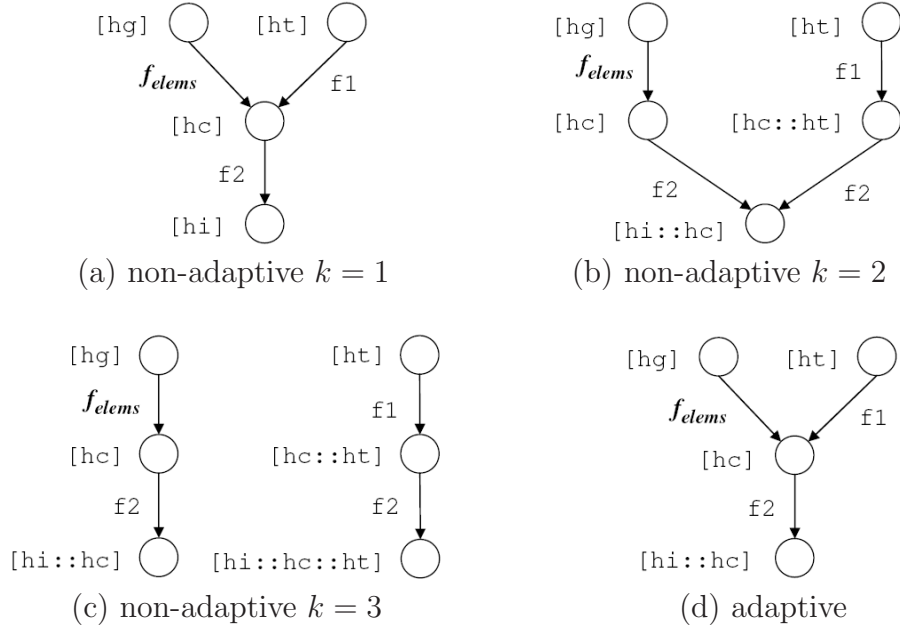


Figure 4.3: Heap abstractions computed for example program.

instances in the above example:

- The algorithm must reason about *must not aliasing relationships* between memory locations, namely, it must infer that statements **p1** and **p2** do not write to the same memory location (and hence are not involved in a race) although both write to elements of integer arrays allocated at the same site **hi** by following field **f2** of **C** objects allocated at the same site **hc**.
- The algorithm must reason about *conditional must not aliasing relationships* between threads/locks and memory locations:
 1. It must correlate threads with the memory locations they access in thread-local structures, namely, it must infer that whenever statement **p1** is executed by distinct threads, it writes to distinct memory locations (and hence there is no race).
 2. It must correlate locks with the memory locations they guard in thread-shared structures, namely, it must infer that whenever statement **p2** is

executed by threads holding distinct locks, it writes to distinct memory locations (and hence there is no race).

We showed in Chapters 2 and 3 that a relatively recent form of may alias analysis for Java called k -object-sensitive analysis by Milanova et al. [60, 61] is effective at approximating aliasing relationships for static race detection. This analysis computes, besides the call graph and points-to information, an abstraction of the heap for a given program. We depict the heap abstraction as a graph with nodes as abstract objects and a labeled directed edge $o_1 \xrightarrow{f} o_2$ whenever field f of an object abstracted by o_1 may point to an object abstracted by o_2 . An abstract object in k -object-sensitive analysis is a sequence $[h_n :: \dots :: h_1]$ where the h_i 's are object allocation sites, h_n and h_1 being the most and least significant, respectively, and the length of the sequence is at most the k value of site h_n which in turn is at least one. Field f is either an instance field or a hypothetical field f_{elems} that is regarded as read/written whenever an array element is read/written; this field is necessary since we do not distinguish between different elements of the same array.

Our earlier race detection algorithm employs a non-adaptive k -object-sensitive analysis that uses the same user-supplied k value for all object allocation sites in the given program. Figures 4.3 (a)–(c) show the heap abstraction it computes for our example program using $k = 1$, $k = 2$, and $k = 3$, respectively. They abstract with increasing precision the data structure shown in Figure 4.2 that is created by the program at run-time. For instance, from Figure 4.2, it is evident that the objects labeled **hc** in the thread-shared structure are different from those labeled **hc** in the thread-local structure. But heap abstraction (a) cannot make this distinction since it abstracts all objects allocated at site **hc** by the same abstract object **[hc]**. Heap abstractions (b) and (c), however, can make this distinction as they abstract objects labeled **hc** in the thread-shared and thread-local structures by **[hc]** and **[hc::ht]**, respectively.

Likewise, from Figure 4.2, it is evident that the objects labeled **hi** in the thread-shared structure are different from those labeled **hi** in the thread-local structure.

Neither heap abstraction (a) nor (b) can make this distinction since each of them abstracts all objects allocated at site `hi` by the same abstract object `[hi]` or `[hi:hc]`, respectively. Only heap abstraction (c) can make this distinction since it abstracts objects labeled `hi` in the thread-shared and thread-local structures by `[hi:hc]` and `[hi:hc:ht]`, respectively. Hence, our earlier algorithm using $k=1$ or $k=2$ reports a false race between statements `p1` and `p2`, namely, it concludes that two different child threads executing them may concurrently write to the same element of the same integer array allocated at site `hi`. Our earlier algorithm using $k=3$, however, does not report a false race between `p1` and `p2`, but it uses the user-supplied k value of three for *all* object allocation sites in the program, when a k value of three is needed just for site `hi` (and a k value of two is needed for site `hc`). In practice, the number of abstract objects grows exponentially with the user-supplied k value. For instance, for our second-largest benchmark `ftp` containing 11,081 sites, the number of abstract objects in the heap abstraction computed by our earlier algorithm using $k=1$ is 11,081, using $k=2$ is 36,040, and using $k=3$ is 364,747, and for our largest benchmark `derby` containing 21,690 sites, our earlier algorithm using $k=2$ (and $k=3$) runs out of memory.

In contrast, our demand-driven algorithm uses an adaptive k -object-sensitive analysis capable of using different k values for different object allocation sites in the same program. It begins with our earlier algorithm using $k=1$ and computes the set R of potential races. It then increments the k values for only those allocation sites of objects on whose fields races are retained in R and repeats the procedure. In successive iterations, false races retained in R due to smaller k values used in previous iterations for those allocation sites get eliminated. The process terminates either when R becomes empty, in which case the program is race-free, or when the k value for none of the allocation sites of objects on whose fields races continue to be retained in R can be increased, in which case it outputs each potential race in R for manual inspection.

We next illustrate our demand-driven algorithm for the above example. In the first iteration, it computes heap abstraction (a) which is the same as that computed by our earlier algorithm using $k=1$. Since this heap abstraction cannot distinguish between objects labeled `hi` in the thread-shared and thread-local structures in Figure 4.2 (it

abstracts all these objects by the same abstract object $[\mathbf{hi}]$), a false race between $\mathbf{p1}$ and $\mathbf{p2}$ is reported in R . Unlike our earlier algorithm using $k=1$ which outputs this false race, our demand-driven algorithm increments the k value of site $\mathbf{hi}$ to two, since that race is deemed to occur on a field (f_{elems}) of abstract object $[\mathbf{hi}]$, and repeats the procedure. In the second iteration, it computes heap abstraction (d) but R still contains the false race since this heap abstraction also abstracts all objects allocated at site $\mathbf{hi}$ by the same abstract object $[\mathbf{hi}:\mathbf{hc}]$. This time, our algorithm increments the k values of both sites $\mathbf{hi}$ and $\mathbf{hc}$ (that is, their k values become three and two respectively), and repeats the procedure. In the third iteration, it computes heap abstraction (c) which is the same as that computed by our earlier algorithm using $k=3$, and R does not contain the false race between $\mathbf{p1}$ and $\mathbf{p2}$ since this heap abstraction distinguishes between objects labeled $\mathbf{hi}$ in the thread-shared and thread-local structures in Figure 4.2 (it abstracts them by different abstract objects $[\mathbf{hi}:\mathbf{hc}]$ and $[\mathbf{hi}:\mathbf{hc}:\mathbf{ht}]$, respectively). In practice, our algorithm uses k values bigger than one for less than 0.01% of all object allocation sites in a given program. For instance, for our largest benchmark **derby** containing 21,690 sites, the heap abstraction computed in the last iteration of our algorithm uses a k value greater than one for only 185 sites, but uses k values as high as eight for those sites.

We have illustrated how k -object-sensitive analysis can be used to statically approximate must not aliasing relationships for race detection, in particular, how it can distinguish between objects labeled $\mathbf{hi}$ in the thread-shared and thread-local structures in Figure 4.2 and thereby prove that statements $\mathbf{p1}$ and $\mathbf{p2}$ executed by two different child threads cannot be involved in a race. It can be used to statically approximate conditional must not aliasing relationships as well, namely, it can be used to prove that:

- From any pair of different objects labeled $\mathbf{ht}$ in the thread-local structure in Figure 4.2, we can only reach different objects labeled $\mathbf{hi}$, from which it is easy to prove that whenever statement $\mathbf{p1}$ is executed by distinct threads, it writes to distinct memory locations, and hence there is no race.
- From any pair of different objects labeled $\mathbf{hc}$ in the thread-shared structure in

Figure 4.2, we can only reach different objects labeled **hi**, from which it is easy to prove that whenever statement **p2** is executed by threads holding distinct locks, it writes to distinct memory locations, and hence there is no race.

Perhaps unintuitively, even $k = 1$ suffices for reasoning about conditional must not aliasing relationships, regardless of how distant the thread objects (labeled **ht**) or lock objects (labeled **hc**) are in the heap from the accessed objects (labeled **hi**). Thus, our earlier algorithm using even $k = 1$ (and hence only a single iteration of our demand-driven algorithm as well) suffices to prove that there is no race in each of the above two cases.

In summary, reasoning about must not aliasing relationships between memory locations in deeply-nested data structures that are created and manipulated by heavily reused code requires high k values for certain sites, but they constitute a tiny fraction of all the sites in the program, whereas a k value of one is sufficient for reasoning about conditional must not aliasing relationships between threads/locks and memory locations. It is this insight that our demand-driven approach exploits to strike a good trade-off between scalability and precision without sacrificing soundness.

4.3 Language

In this section, we present a language for formalizing k -object-sensitive analysis and our demand-driven race detection algorithm. The abstract syntax of the language is shown in Figure 4.4. A program consists of a set of methods $\mathbb{M} = \mathbb{M}_{stat} \uplus \mathbb{M}_{inst}$ where $\mathbb{M}_{stat}$ includes all static methods and $\mathbb{M}_{inst}$ includes all instance methods. The difference between a static and an instance method is that the latter has an implicit 0^{th} formal argument, called **this** in Java. We presume that $\mathbb{M}_{stat}$ includes a distinguished main method denoted m_{main} at which the program begins execution.

Each method declares a set of local variables which also includes its formal arguments and return/throw variables (exception handling code in Java is compiled away). We use $\mathbb{V}$ to denote the set of all local variables in all methods. Function $margin_0$ maps each instance method to its 0^{th} **this** argument while functions $margin$ and

m	$\in$	$\mathbb{M} = \mathbb{M}_{stat} \uplus \mathbb{M}_{inst}$	(method)
v	$\in$	$\mathbb{V}$	(local variable)
f	$\in$	$\mathbb{F}$	(instance field)
g	$\in$	$\mathbb{G}$	(static field)
p	$\in$	$\mathbb{P}$	(program point)
h	$\in$	$\mathbb{H}$	(object allocation site)
i	$\in$	$\mathbb{I} = \mathbb{I}_{stat} \uplus \mathbb{I}_{inst}$	(call site)
s	$\in$	$\mathbb{S}$	(statement)

$s ::=$	$v_1 = v_2$	$ $	$v = \mathbf{new} \ h$	$ $	$v_2 = v_1.f$	$ $	$v_1.f = v_2$	$ $
	$v = g$	$ $	$g = v$	$ $	i_{stat}	$ $	i_{inst}	

$\mathbb{N}^+$	$=$	$\{ 1, 2, 3, \dots \}$	
$marg_0$	$:$	$\mathbb{M}_{inst} \rightarrow \mathbb{V}$	(this arg of method)
$marg, mret$	$:$	$(\mathbb{M} \times \mathbb{N}^+) \multimap \mathbb{V}$	(args/rets of method)
$meth$	$:$	$\mathbb{P} \rightarrow \mathbb{M}$	(containing method of point)
$stmt$	$:$	$\mathbb{P} \rightarrow \mathbb{S}$	(statement at point)
$iarg_0$	$:$	$\mathbb{I}_{inst} \rightarrow \mathbb{V}$	(this arg of call site)
$iarg, iret$	$:$	$(\mathbb{I} \times \mathbb{N}^+) \multimap \mathbb{V}$	(args/rets of call site)
$trgt$	$:$	$(\mathbb{I}_{stat} \rightarrow \mathbb{M}_{stat}) \uplus$ $(\mathbb{I}_{inst} \times \mathbb{H}) \multimap \mathbb{M}_{inst}$	(method resolution)

Figure 4.4: Abstract syntax.

mret provide the formal arguments (other than `this`) and return/throw variables, respectively, of each method.

We use $\mathbb{F}$ to denote the set of all instance fields. Both k -object-sensitive analysis and our race detection algorithm does not distinguish between different elements of the same array; hence, $\mathbb{F}$ includes a hypothetical field f_{elems} that is treated as read/written whenever an array element is read/written. We use $\mathbb{G}$ to denote the set of all static fields, which are akin to global variables.

We use $\mathbb{P}$ to denote the set of all program points in all methods. Function *meth* provides the containing method of each point and function *stmt* provides the statement at each point, which may be a copy of a local variable $v_1 = v_2$, an object allocation $v = \text{new } h$, a heap read $v_2 = v_1.f$, a heap write $v_1.f = v_2$, a read of a global variable $v = g$, a write to a global variable $g = v$, a static method call i_{stat} , or an instance method call i_{inst} .

We use $\mathbb{H}$ to denote the set of all object allocation sites and $\mathbb{I} = \mathbb{I}_{stat} \uplus \mathbb{I}_{inst}$ to denote the set of all call sites, where $\mathbb{I}_{stat}$ contains calls to static methods and $\mathbb{I}_{inst}$ contains calls to instance methods. Functions *iarg₀*, *iarg*, and *iret* provide actual arguments and return/throw variables of call sites analogously to functions *marg₀*, *marg*, and *mret* which provide formal arguments and return/throw variables of methods. Function *trgt* performs method resolution. The target method of a call site i_{stat} is uniquely determined statically given the call site itself whereas the target method of a call site i_{inst} is uniquely determined statically given both the call site and the object allocation site of i_{inst} 's 0th argument.

Our language elides operations on primitive types (e.g., boolean, int, etc.) as well as concurrency constructs such as operations for acquiring/releasing locks since k -object-sensitive analysis treats them as no-ops and the presentation of our race detection approach does not require them.

It is easy to transform the program in our running example from Figure 4.1 into our language. We have $m_{main} = \text{T.main}$, $\mathbb{M}_{stat} = \{ \text{T.main}, \text{C.newInstance}, \dots \}$, $\mathbb{M}_{inst} = \{ \text{T.<init>}, \text{T.run}, \text{C.<init>}, \dots \}$ where `<init>` denotes a constructor, $\mathbb{F} = \{ f_{elems}, \text{T.f1}, \text{C.f2}, \dots \}$, $\mathbb{G} = \{ \text{T.g}, \dots \}$, and $\mathbb{H} = \{ \text{hg}, \text{ht}, \text{hc}, \text{hi}, \dots \}$. Note that Figure 4.1 only shows the *application code* for the example; the ellipses in the

n	$\in \mathbb{N}^+$	(k value)
o	$\in \mathbb{O} = \mathbb{H}^n$	(abstract object)
c	$\in \mathbb{C} = \{\epsilon\} \cup \mathbb{O}$	(abstract context)
Σ	$: \mathbb{H} \rightarrow \mathbb{N}^+$	(k value map)
R	$\subseteq \mathbb{C} \times \mathbb{P}$	(reachable code)
P_V	$\subseteq \mathbb{C} \times \mathbb{V} \times \mathbb{O}$	(points-to info for local variables)
P_G	$\subseteq \mathbb{G} \times \mathbb{O}$	(points-to info for static fields)
H	$\subseteq \mathbb{O} \times \mathbb{F} \times \mathbb{O}$	(heap abstraction)
G	$\subseteq \mathbb{C} \times \mathbb{I} \times \mathbb{C} \times \mathbb{M}$	(call graph)

Figure 4.5: Analysis domains.

above sets indicate that they contain additional elements arising from *library code* not shown in the figure.

4.4 k -Object-Sensitive Analysis

In this section, we present our implementation of k -object-sensitive analysis. Figure 4.5 presents the analysis domains. The analysis is *object sensitive*, that is, it abstracts different objects allocated at the same site by potentially different abstract objects. An abstract object o is a non-empty sequence of object allocation sites of the form $[h_n :: \dots :: h_1]$. We call h_n and h_1 the most and least significant sites, respectively. We use $o.\mathbf{car}$ to denote the head h_n and $o.\mathbf{cdr}$ to denote the tail $[h_{n-1} :: \dots :: h_1]$. Then, different objects allocated at site h_n are abstracted by different abstract objects o_1 and o_2 iff $o_1.\mathbf{car} = o_2.\mathbf{car} = h_n$ and $o_1.\mathbf{cdr} \neq o_2.\mathbf{cdr}$. We use $\mathbb{O}$ to denote the set of all abstract objects.

The analysis is *context sensitive*, that is, it analyzes each method in potentially multiple abstract contexts. An abstract context is either a distinguished context ϵ which denotes the sole context in which the main method m_{main} is analyzed or it is an abstract object. We use $\mathbb{C}$ to denote the set of all abstract contexts.

The analysis is parameterized by a function Σ which maps each object allocation site in $\mathbb{H}$ to a positive integer which we call its k value. This parameterization allows us to uniformly express both non-adaptive and adaptive k -object-sensitive analysis:

a non-adaptive analysis uses the same k value for all sites whereas an adaptive one can use different k values for different sites.

The analysis computes sets P_V , P_G , H , and G where:

- P_V , the points-to information for local variables, contains each tuple (c, v, o) such that local variable v may point to abstract object o in abstract context c of v 's declaring method. Note that the points-to information is both context and object sensitive.
- P_G , the points-to information for static fields, contains each tuple (g, o) such that static field g may point to abstract object o . Note that the points-to information is object sensitive but not context sensitive: static fields in Java are akin to global variables that are declared outside of all methods, and abstract contexts are associated only with methods.
- H , the heap abstraction, contains each tuple (o_1, f, o_2) such that instance field f (or the hypothetical field f_{elems}) of abstract object o_1 may point to abstract object o_2 .
- G , the call graph, contains each tuple (c_1, i, c_2, m) such that call site i in abstract context c_1 of its containing method may call method m in abstract context c_2 . Note that the call graph is context sensitive.

Besides the form of parameterization manifested in Σ , Milanova et al. [60, 61] present several dimensions to parameterize k -object-sensitive analysis, e.g., the points-to information for certain local variables may be context insensitive, that is, P_V may contain tuples of the form (v, o) instead of (c, v, o) . One of the challenges in devising a client (like a static race detector) of an alias analysis which provides many forms of parameterization (like k -object-sensitive analysis) lies in choosing from those forms of parameterization and devising a technique that automatically instantiates the chosen form of parameterization (e.g., a technique that decides what k value to use for each object allocation site for the form of parameterization manifested in Σ) in a manner that makes the client effective in practice. We have chosen only the form of parameterization provided in Σ for our static race detection algorithm. We present

$\Sigma \vdash (R, P_V, P_G, H, G)$ iff:

$$\text{meth}(p) = m_{\text{main}} \Rightarrow (\epsilon, p) \in R \quad (4.1)$$

$$((c, p) \in R \wedge \text{stmt}(p) \equiv v_1 = v_2 \wedge (c, v_2, o) \in P_V) \Rightarrow (c, v_1, o) \in P_V \quad (4.2)$$

$$((c, p) \in R \wedge \text{stmt}(p) \equiv v = \text{new } h) \Rightarrow (c, v, h \oplus_{\Sigma(h)} c) \in P_V \quad (4.3)$$

$$((c, p) \in R \wedge \text{stmt}(p) \equiv v_2 = v_1.f \wedge (c, v_1, o_1) \in P_V \wedge (o_1, f, o_2) \in H) \Rightarrow (c, v_2, o_2) \in P_V \quad (4.4)$$

$$((c, p) \in R \wedge \text{stmt}(p) \equiv v_1.f = v_2 \wedge (c, v_1, o_1) \in P_V \wedge (c, v_2, o_2) \in P_V) \Rightarrow (o_1, f, o_2) \in H \quad (4.5)$$

$$((c, p) \in R \wedge \text{stmt}(p) \equiv v = g \wedge (g, o) \in P_G) \Rightarrow (c, v, o) \in P_V \quad (4.6)$$

$$((c, p) \in R \wedge \text{stmt}(p) \equiv g = v \wedge (c, v, o) \in P_V) \Rightarrow (g, o) \in P_G \quad (4.7)$$

$$\begin{aligned} & ((c, p_1) \in R \wedge \text{stmt}(p_1) \equiv i_{\text{stat}} \wedge \text{trgt}(i_{\text{stat}}) = m) \Rightarrow \\ & \left(\begin{array}{l} (1) \text{ meth}(p_2) = m \Rightarrow (c, p_2) \in R \\ \wedge (2) (c, \text{iarg}(i_{\text{stat}}, n), o) \in P_V \Rightarrow (c, \text{marg}(m, n), o) \in P_V \\ \wedge (3) (c, \text{mret}(m, n), o) \in P_V \Rightarrow (c, \text{iret}(i_{\text{stat}}, n), o) \in P_V \\ \wedge (4) (c, i_{\text{stat}}, c, m) \in G \end{array} \right) \end{aligned} \quad (4.8)$$

$$\begin{aligned} & \left(\begin{array}{l} (a) (c, p_1) \in R \\ \wedge (b) \text{ stmt}(p_1) \equiv i_{\text{inst}} \\ \wedge (c) (c, \text{iarg}_0(i_{\text{inst}}), o_0) \in P_V \\ \wedge (d) \text{ trgt}(i_{\text{inst}}, o_0, \mathbf{car}) = m \end{array} \right) \Rightarrow \\ & \left(\begin{array}{l} (1) \text{ meth}(p_2) = m \Rightarrow (o_0, p_2) \in R \\ \wedge (2) (o_0, \text{marg}_0(m), o_0) \in P_V \\ \wedge (3) (c, \text{iarg}(i_{\text{inst}}, n), o) \in P_V \Rightarrow (o_0, \text{marg}(m, n), o) \in P_V \\ \wedge (4) (o_0, \text{mret}(m, n), o) \in P_V \Rightarrow (c, \text{iret}(i_{\text{inst}}, n), o) \in P_V \\ \wedge (5) (c, i_{\text{inst}}, o_0, m) \in G \end{array} \right) \end{aligned} \quad (4.9)$$

Figure 4.6: k -object-sensitive analysis.

a technique that automatically instantiates Σ in Section 4.5 and demonstrate its effectiveness on static race detection for real-world multithreaded Java programs in Section 4.6.

Before presenting the k -object-sensitive analysis itself, we define how an abstract object o is built given an abstract context c , an object allocation site h , and a k value n (recall that k values are positive integers, that is, $n \geq 1$).

$$o = h \oplus_n c \text{ iff: } \left(\begin{array}{l} |o| = \min(n, |c| + 1) \wedge o.\mathbf{car} = h \wedge \\ \forall n' \in [1..|o| - 1] : o.\mathbf{cdr}^{n'}.\mathbf{car} = c.\mathbf{cdr}^{n'-1}.\mathbf{car} \end{array} \right)$$

Intuitively, we say $o = h \oplus_n c$ iff o is a (non-empty) sequence whose head is h and whose tail comprises at most the $n - 1$ most significant sites in c in order.

Our k -object-sensitive analysis is presented in Figure 4.6. The judgment $\Sigma \vdash (R, V_P, V_G, H, G)$ holds if (R, V_P, V_G, H, G) is the least solution of Rules (4.1)–(4.9). R is an intermediate set containing each tuple (c, p) such that program point p may be reachable in context c of its containing method, while V_P , V_G , H , and G are output sets described above. We explain the rules using our running example from Figure 4.1. From Rule (4.1), the analysis infers that each program point p in method **T.main** may be reachable in context ϵ . Then, one of Rules (4.2)–(4.9) is applied for each such p , depending upon the kind of statement at p .

Rule (4.3) says that if statement $v = \mathbf{new} \ h$ may be reachable in context c , then v may point to abstract object $h \oplus_n c$, where n is the k value of h , that is, $n = \Sigma(h)$. Method **T.main**, deemed reachable in context ϵ , contains two such statements, namely, **T.g = new C[*]** labeled **hg** and **t = new C()** labeled **ht**. From the former, the analysis infers that static field **T.g** may point to abstract object **[hg]** since **hg** $\oplus_n \epsilon = \mathbf{[hg]}$ for any k value n . Likewise, from the latter, the analysis infers that local variable **t** may point to abstract object **[ht]**.

Rule (4.9) says that if call site i_{inst} may be reachable in context c (items (a) and (b)) and its 0^{th} argument may point to abstract object o_0 in context c (item (c)) and m is the target instance method of i_{inst} when the object allocation site of i_{inst} 's 0^{th} argument is $o_0.\mathbf{car}$ (item (d)), then each program point in m may be

reachable in context o_0 (item (1)) and, moreover, the **this** argument of m may point to abstract object o_0 in that context (item (2)). The rule also propagates points-to information from the remaining arguments of i_{inst} to those of m (item (3)) and from return/throw variables of m to those of i_{inst} (item (4)), and reflects the call in the call graph (item (5)). For our example, method **T.main**, deemed reachable in context ϵ , contains two call sites to instance methods, namely, the calls to **C.<init>** and **T.run** in the body of the second loop (strictly, the second call is to the **start** method of class `java.lang.Thread` but for brevity, we regard it as a call to **T.run** as the **start** method simply calls **T.run**). Also, the analysis has already inferred that the 0th argument of both calls, local variable **t**, may point to abstract object **[ht]** (see above). Hence, it next infers that each program point in **C.<init>** and **T.run** may be reachable in context **[ht]**, and the **this** argument of those methods may point to abstract object **[ht]** in that context. The special treatment of the **this** argument in Rule (4.9) is a hallmark of k -object-sensitive analysis that exploits object-oriented idioms in Java for precision. An instance method is analyzed in a separate context for each abstract object to which its **this** argument may point, and in that context, the **this** argument points to only one abstract object.

Static methods, however, do not possess a **this** argument. A key difference between our implementation of k -object-sensitive analysis and Milanova et al.’s lies in determining the contexts in which such methods are analyzed. Milanova et al. claim that analyzing static methods in multiple contexts leads to small gains in precision for their clients (side-effect analysis and call graph construction) while making their analysis more complex, and they therefore analyze each static method in a single context ϵ . In our experiments, however, this is a significant source of imprecision (note that in our case the client is race detection). For instance, for our example, objects of class **C** are created by calling the public static method **C.newInstance** since **C.<init>** is private, a pattern commonly used in Java. If **C.newInstance** is analyzed in a single context ϵ , then all objects allocated at site **hc** occurring in **C.newInstance** are abstracted by the single abstract object **[hc]** (since $hc \oplus_n \epsilon = [hc]$ for any k value n), which causes Rule (4.5) applied to writes **T.g[i] = C.newInstance()** and **this.f1 = C.newInstance()** to create the edges labeled f_{elems} and **f1**, respectively,

that have the same target `[hc]` in heap abstraction (a) in Figure 4.3. Moreover, it causes method `C.<init>`, called from `C.newInstance`, to be analyzed in a single context `[hc]` as the 0^{th} argument of that call site points to the single abstract object `[hc]`. Finally, all objects allocated at site `hi` occurring in `C.<init>` are abstracted by the same abstract object `[hi]` or `[hi::hc]` depending upon whether the k value of that site is one or greater than one, respectively, and Rule (4.5) applied to the write `this.f2 = new int[*]` creates the single edge labeled `f2` in heap abstraction (a). Recall from Section 4.2 that this heap abstraction causes a false race to be reported between statements `p1` and `p2`.

In contrast, our implementation analyzes each static method in potentially multiple contexts, namely, Rule (4.8) causes each static method to be analyzed in each context of each of its callers. For our example, this causes method `C.newInstance`, called from methods `T.main` and `T.<init>`, to be analyzed in contexts ϵ and `[ht]` since the analysis has inferred that `T.main` and `T.<init>` may be reachable in contexts ϵ and `[ht]`, respectively (see above). Then, if the k value of site `hc` occurring in `C.newInstance` is at least two, objects allocated at that site are abstracted by one of abstract objects `[hc]` and `[hc::ht]`, and method `C.<init>` is analyzed in contexts `[hc]` and `[hc::ht]`. Finally, if the k value of site `hi` occurring in `C.<init>` is at least three, then objects allocated at that site are abstracted by one of abstract objects `[hi::hc]` and `[hi::hc::ht]`, yielding heap abstraction (c) in Figure 4.3. Recall from Section 4.2 that this heap abstraction does not report a false race between `p1` and `p2`.

Another novel idea in our implementation is the pre-computation of a set $\mathcal{A}$ that is a close over-approximation of both sets $\mathbb{C}$ and $\mathbb{O}$ of abstract contexts and abstract objects, respectively, that will be used by the analysis. For scalability, our k -object-sensitive analysis is expressed and solved using `bddbddb` [87], a Binary Decision Diagram (BDD) based Datalog solver, in particular, the rules in Figure 4.6 are expressed as Datalog rules and all relations, including input relations that encode basic program facts such as `stmt` and `trgt`, temporary relations such as R , and output relations such as P_V , P_G , H , and G are represented as BDDs. BDDs are particularly effective at compactly representing and efficiently manipulating relations with high

$\Sigma \vdash \text{ctxts}$ iff:

$$\epsilon \in \text{ctxts}(m_{\text{main}}) \quad (4.10)$$

$$\left(\begin{array}{l} (a) \ c \in \text{ctxts}(m_1) \\ \wedge (b) \ \text{meth}(p) = m_1 \\ \wedge (c) \ \text{stmt}(p) \equiv i_{\text{stat}} \\ \wedge (d) \ \text{trgt}(i_{\text{stat}}) = m_2 \end{array} \right) \Rightarrow c \in \text{ctxts}(m_2) \quad (4.11)$$

$$\left(\begin{array}{l} (a) \ c \in \text{ctxts}(m_1) \\ \wedge (b) \ \text{meth}(p) = m_1 \\ \wedge (c) \ \text{stmt}(p) \equiv v = \text{new } h \\ \wedge (d) \ (iarg_0(i_{\text{inst}}, h) \in P'_V) \\ \wedge (e) \ \text{trgt}(i_{\text{inst}}, h) = m_2 \end{array} \right) \Rightarrow (h \oplus_{\Sigma(h)} c) \in \text{ctxts}(m_2) \quad (4.12)$$

$\Sigma \vdash \mathcal{A}$ iff:

$$\epsilon \in \mathcal{A} \quad (4.13)$$

$$(c \in \text{ctxts}(m) \wedge \text{meth}(p) = m \wedge \text{stmt}(p) \equiv v = \text{new } h) \Rightarrow (h \oplus_{\Sigma(h)} c) \in \mathcal{A} \quad (4.14)$$

Figure 4.7: Computation of $\mathcal{A}$.

levels of redundancy, such as those arising in context sensitive, whole-program analyses. BDDs, however, require relations over integer domains. It is straightforward to map each element in a domain like $\mathbb{H}$ (or $\mathbb{M}$, $\mathbb{V}$, $\mathbb{F}$, etc.), that is linear in the size of the program, to an integer in the range $[0..|\mathbb{H}| - 1]$ but it is not so for domains $\mathbb{O} = \mathbb{H}^n$ and $\mathbb{C} = \{\epsilon\} \cup \mathbb{O}$ since it is not practical to enumerate each element in $\mathbb{H}^n$ for $n > 1$.

Our insight is that we need to map to integers only those elements in $\mathbb{H}^n$ that will be used by the analysis, in particular, we only need to map to integers elements in a subset of $\mathbb{H}^n$ such that whenever the Datalog solver executes Rule (4.3) in Figure 4.6 while performing the analysis for a given program and a given Σ , the element $h \oplus_{\Sigma(h)} c$ is guaranteed to belong to that subset. Now, it is not possible to compute the subset exactly (that is, the subset contains an element in $\mathbb{H}^n$ iff it will be used by the analysis) without performing the analysis itself. We therefore compute an over-approximation $\mathcal{A}$ of the subset (that is, if an element in $\mathbb{H}^n$ will be used by the analysis then $\mathcal{A}$ contains it).

The computation of $\mathcal{A}$ is shown in Figure 4.7. The first two rules compute the least function $ctxts$ such that $ctxts(m)$ is an over-approximation of the set of contexts in which method m will be analyzed during the analysis. Rule (4.10) is the base case and states that m_{main} will be analyzed in context ϵ . Rules (4.11) and (4.12), the inductive cases, are mutually recursive and estimate the contexts in which static methods (besides m_{main}) and instance methods may be analyzed, respectively. They are analogous to Rules (4.8) and (4.9) from Figure 4.6. The key difference is that Rule (4.9) uses context and object sensitive points-to information $P_V \subseteq (\mathbb{C} \times \mathbb{V} \times \mathbb{O})$ computed during k -object-sensitive analysis itself whereas Rule (4.12) uses context and object insensitive points-to information $P'_V \subseteq (\mathbb{V} \times \mathbb{H})$ computed by a 0-CFA-based analysis which also constructs the initial context insensitive call graph of the given program. The k -object-sensitive analysis essentially refines the 0-CFA-based analysis by computing context and object sensitive points-to information P_V and a context sensitive call graph G .

Finally, $\mathcal{A}$ is computed as the smallest set satisfying Rules (4.13) and (4.14). It is an over-approximation of both $\mathbb{C}$ and $\mathbb{O}$, in particular, Rule (4.13) states that $\mathcal{A}$ contains ϵ since clearly this element is used during the analysis (see Rule (4.1) in Figure 4.6) while Rule (4.14) states that if a method m may be analyzed in context c and m contains an object allocation site h , then $\mathcal{A}$ contains $h \oplus_{\Sigma(h)} c$ as Rule (4.3) in Figure 4.6 will use this element if m is indeed deemed reachable in context c during the analysis.

4.5 Demand-Driven Race Detection Algorithm

In this section, we present our demand-driven race detection algorithm. The algorithm is shown in Figure 4.8. The outermost loop first performs the k -object-sensitive analysis presented in Section 4.4, denoted by function `kObjectSensitiveAnalysis`. Recall that the analysis is parameterized by a function Σ mapping each object allocation site in the given program to its k value; in the first iteration, our algorithm uses $\Sigma = \lambda h.1$. In addition to Σ , function `kObjectSensitiveAnalysis` takes inputs (not shown in the figure) representing facts about the given Java program

```

var  $\Sigma : \mathbb{H} \rightarrow \mathbb{N}^+ = \lambda h.1$ 
var ptsG : set of  $(\mathbb{G} \times \mathcal{A})$ 
var heap : set of  $(\mathcal{A} \times \mathbb{F} \times \mathcal{A})$ 
var cscg : set of  $(\mathcal{A} \times \mathbb{I} \times \mathcal{A} \times \mathbb{M})$ 
var ptsV : set of  $(\mathcal{A} \times \mathbb{P} \times \mathbb{V} \times \mathcal{A})$ 
var esc0 : set of  $(\mathcal{A} \times \mathbb{P} \times \mathcal{A})$ 
var originalRaces, parallelRaces, unlockedRaces,
    ultimateRaces : set of  $(\mathcal{A} \times \mathcal{A} \times \mathbb{P} \times \mathcal{A} \times \mathcal{A} \times \mathbb{P})$ 
var ultimateRacesWithObject :
    set of  $(\mathcal{A} \times \mathcal{A} \times \mathbb{P} \times \mathcal{A} \times \mathcal{A} \times \mathbb{P} \times \mathcal{A})$ 
var A : set of  $\mathcal{A} = \emptyset$ 
var H : set of  $\mathbb{H}$ 
do
  (ptsG, heap, cscg) = kObjectSensitiveAnalysis( $\Sigma, \dots$ );
  originalRaces = computeOriginalRaces(cscg, ...);
  (ptsV, esc0) = mayAliasAndThreadEscapeAnalysis(
    ptsG, heap, cscg, ...);
  parallelRaces = computeParallelRaces(
    originalRaces, cscg, ...);
  unlockedRaces = computeUnlockedRaces(
    originalRaces, cscg, ...);
  ultimateRacesWithObject = computeUltimateRaces(
    ptsV, esc0, parallelRaces, unlockedRaces);
  H =  $\emptyset$ ;
  for each  $[h_n :: \dots :: h_1] \in \pi_7(\text{ultimateRacesWithObject})$  do
    if  $[h_n :: \dots :: h_1] \notin A$  then
      A = A  $\cup \{[h_n :: \dots :: h_1]\}$ 
      if  $n = |\{h_1, \dots, h_n\}|$  then
        H = H  $\cup \{h_1, \dots, h_n\}$ 
    for each  $h \in H$  do
       $\Sigma = \Sigma[h \mapsto (\Sigma(h) + 1)]$ 
while H  $\neq \emptyset$ 
ultimateRaces =  $\pi_{1,2,3,4,5,6}(\text{ultimateRacesWithObject})$ 
output each race in ultimateRaces

```

Figure 4.8: Demand-driven static race detection algorithm.

such as functions *stmt* and *trgt* as well as the context insensitive points-to information and context insensitive call graph computed by a 0-CFA-based analysis. The output of `kObjectSensitiveAnalysis` includes sets `ptsG`, `heap`, and `cscg`, denoting object sensitive points-to information for static fields, the heap abstraction, and the context sensitive call graph, that is, sets P_G , H , and G from Section 4.4.

The algorithm next computes an initial over-approximation of the set of races in the program, denoted `originalRaces`, using function `computeOriginalRaces`. The set contains each tuple of the form $(c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2)$ such that the statements at program points p_1 and p_2 may be involved in a race in abstract contexts c_1^m and c_2^m of their containing methods, respectively, when executed by abstract threads c_1^t and c_2^t , respectively. An abstract thread is the abstract context of the thread's starting method. In the case of the implicit main thread, the starting method is m_{main} , and the abstract thread is ϵ (recall that m_{main} is analyzed in the sole context ϵ). In the case of any explicit child thread, the starting method is the `start` method of class `java.lang.Thread`, and the abstract thread is some abstract context of the `start` method (it is never ϵ since this method is an instance method).

Then, $(c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in \text{originalRaces}$ if the containing method of p_1 (resp. p_2) may be reachable in abstract context c_1^m (resp. c_2^m) in the context sensitive call graph `cscg` from its thread's starting method in abstract context c_1^t (resp. c_2^t). An additional condition that must be satisfied is that the statements at p_1 and p_2 must both access the same field and at least one of them must be a write (Java's semantics ensures that they cannot be involved in a race otherwise). Thus, we must have either of the following:

- $stmt(p_1) \equiv v'_1 = v_1.f$ or $v_1.f = v'_1$ and $stmt(p_2) \equiv v_2.f = v'_2$.
- $stmt(p_1) \equiv v_1 = g$ or $g = v_1$ and $stmt(p_2) \equiv g = v_2$.

For our running example from Figure 4.1, we focus on the following three tuples to illustrate our race detection algorithm:

T1: ([ht], [ht], p1, [ht], [ht], p1)
 T2: ([ht], [ht], p2, [ht], [ht], p2)
 T3: ([ht], [ht], p1, [ht], [ht], p2)

Program points **p1** and **p2** both occur in method **T.run** which may be executed by abstract thread **[ht]** in abstract context **[ht]**. Also, the statements at both **p1** and **p2** write to array elements, that is, to the same field f_{elems} . Thus, each of these tuples satisfies the conditions necessary for inclusion in **originalRaces**. Note, however, that all three tuples are false races.

The algorithm next performs a combined may alias and thread escape analysis, denoted by function **mayAliasAndThreadEscapeAnalysis**, that computes set **esc0** containing each tuple (c, p, o) such that abstract object o may escape the current thread just before program point p in abstract context c of p 's containing method, and set **ptsV** containing each tuple (c, p, v, o) such that local variable v may point to abstract object o just before program point p in abstract context c of p 's containing method. The function takes as input besides **cscg**, sets **ptsG** and **heap** computed by k -object-sensitive analysis: the combined may alias and thread escape analysis falls back upon the flow insensitive information in **ptsG** for reads from static fields (which, being global variables, are globally thread-escaping) and in **heap** for reads from instance fields of abstract objects it deems as possibly thread-escaping.

The algorithm next performs a may-happen-in-parallel analysis, denoted by function **computeParallelRaces**, which outputs set **parallelRaces** containing each tuple $(c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in \mathbf{originalRaces}$ such that the program's thread structure does not prevent abstract threads c_1^t and c_2^t from simultaneously executing the statements at p_1 and p_2 in abstract contexts c_1^m and c_2^m , respectively. Tuples not retained in **parallelRaces** are race-free. For our running example, tuples T1, T2, and T3 are all retained in **parallelRaces** since the program's thread structure allows the pair of statements in each of them to execute simultaneously.

The algorithm next performs conditional must not alias analysis, denoted by function **computeUnlockedRaces**, which outputs set **unlockedRaces** containing each tuple $(c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in \mathbf{originalRaces}$ such that the program's lock structure does

not prevent abstract threads c_1^t and c_2^t from simultaneously executing the statements at p_1 and p_2 in abstract contexts c_1^m and c_2^m , respectively. Tuples not retained in **unlockedRaces** are race-free. Returning to our running example, tuples T1 and T2 are not retained in **unlockedRaces** even in the first iteration of our algorithm because a k value of one suffices to reason about conditional must not aliasing, namely, T1 is race-free since whenever the statement at **p1** is executed by distinct threads, it writes to distinct locations, and T2 is race-free since whenever the statement at **p2** is executed by threads holding distinct locks, it writes to distinct locations.

Suppose the partial function $base : \mathbb{P} \rightarrow \mathbb{V}$ is defined such that if $stmt(p)$ is of the form $v_2 = v_1.f$ or $v_1.f = v_2$ then $base(p) = v_1$. Then, our earlier algorithm computes the following sets besides **parallelRaces** and **unlockedRaces**:

$$\begin{aligned} \text{aliasingRaces} &= \{ (c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in \text{originalRaces} \mid \\ &\quad (base(p_1) = v_1 \wedge base(p_2) = v_2) \Rightarrow \\ &\quad (\exists o : (c_1^m, p_1, v_1, o) \in \text{ptsV} \wedge (c_2^m, p_2, v_2, o) \in \text{ptsV}) \} \\ \text{escapingRaces} &= \{ (c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in \text{originalRaces} \mid \\ &\quad (base(p_1) = v_1 \wedge base(p_2) = v_2) \Rightarrow \\ &\quad ((\exists o_1 : (c_1^m, p_1, v_1, o_1) \in \text{ptsV} \wedge (c_1^m, p_1, o_1) \in \text{esc0}) \wedge \\ &\quad (\exists o_2 : (c_2^m, p_2, v_2, o_2) \in \text{ptsV} \wedge (c_2^m, p_2, o_2) \in \text{esc0})) \} \end{aligned}$$

aliasingRaces contains each tuple such that both statements in it may access the same location and **escapingRaces** contains each tuple such that each statement in it may access a thread-escaping location. A tuple not retained in **aliasingRaces** or **escapingRaces** is race-free. For our running example, tuples T1, T2, and T3 are all retained in both **aliasingRaces** and **escapingRaces** because we have:

$$\begin{aligned} ([ht], p1, v2, [hi]) &\in \text{ptsV} & ([ht], p1, [hi]) &\in \text{esc0} \\ ([ht], p2, v5, [hi]) &\in \text{ptsV} & ([ht], p2, [hi]) &\in \text{esc0} \end{aligned}$$

that is, local variables **v2** and **v5** may point to abstract object **[hi]** just before program points **p1** and **p2**, respectively, in abstract context **[ht]**, and moreover, abstract object **[hi]** may be thread-escaping just before program points **p1** and **p2** in

abstract context `[ht]`. Our earlier algorithm then computes the final set of potential races as:

$$\begin{aligned} \text{ultimateRaces} = & \text{aliasingRaces} \cap \text{escapingRaces} \cap \\ & \text{parallelRaces} \cap \text{unlockedRaces} \end{aligned}$$

and terminates after reporting each race in `ultimateRaces`. For our running example, it reports tuple T3 (recall that tuples T1 and T2 are not retained in `unlockedRaces`).

In contrast, our demand-driven algorithm uses `ultimateRaces` to potentially refine Σ and execute another iteration of the outermost loop in Figure 4.8 to yield a potentially smaller set of races to be reported. The smaller set is sound but more precise in that it eliminates false races that are deemed to occur in the previous iteration due to smaller k values used in the Σ in that iteration for the allocation sites of objects on whose fields those races are deemed to occur. To refine Σ , we compute the following set which makes explicit the abstract objects on whose fields the races are deemed to occur:

$$\begin{aligned} \text{ultimateRacesWithObject} = & \{ (c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2, o) \mid \\ & (c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in (\text{parallelRaces} \cap \text{unlockedRaces}) \\ & \wedge \left(\begin{array}{l} \text{if } (base(p_1) = v_1 \wedge base(p_2) = v_2) \text{ then} \\ \quad ((c_1^m, p_1, v_1, o) \in \text{ptsV} \wedge (c_1^m, p_1, o) \in \text{esc0} \wedge \\ \quad (c_2^m, p_2, v_2, o) \in \text{ptsV} \wedge (c_2^m, p_2, o) \in \text{esc0}) \\ \text{else } o = \epsilon \end{array} \right) \} \end{aligned}$$

This set essentially contains each tuple $(c_1^t, c_1^m, p_1, c_2^t, c_2^m, p_2) \in \text{ultimateRaces}$ along with each abstract object o such that:

- If the statements at p_1 and p_2 access the same instance field or array elements, then $base(p_1) = v_1$ and $base(p_2) = v_2$, in which case o is an abstract object such that both v_1 and v_2 may point to o just before p_1 and p_2 in abstract contexts c_1^m and c_2^m , respectively, and moreover, o may be thread-escaping just before p_1 and p_2 in abstract contexts c_1^m and c_2^m , respectively.
- If the statements at p_1 and p_2 access the same static field, then $base(p_1)$ and

$base(p_2)$ are undefined and o does not exist, but for the sake of uniformity, we use $o = \epsilon$.

For our running example, the above set contains the lone tuple T3 in `ultimateRaces` along with abstract object `[hi]`.

Our algorithm next computes the set H of object allocation sites whose k values must be incremented in the refined Σ to be used in the next iteration as follows. Suppose `ultimateRacesWithObject` contains a tuple manifesting a potential race on an instance field or array element of abstract object o , in which case o is a non-empty sequence of object allocation sites of the form $[h_n :: \dots :: h_1]$. For termination, the algorithm must check for two conditions:

1. o must not have been considered in a previous iteration (denoted by the check $[h_n :: \dots :: h_1] \notin A$).
2. o must not contain duplicate sites, that is, sites $h_1, \dots, h_n$ must be distinct (denoted by the check $n = |\{h_1, \dots, h_n\}|$).

If both these conditions hold, the algorithm adds each of sites $h_1, \dots, h_n$ to H . After performing the above two checks for each tuple in `ultimateRacesWithObject`, the algorithm refines Σ by incrementing the k value for each site in the resulting H , and repeats the entire procedure if H is non-empty. If H is empty, it means that the algorithm is unable to increment the k value for any of the allocation sites of objects on whose fields races are deemed to occur. At this point, it projects away the abstract object in each tuple in `ultimateRacesWithObject` and reports the potential race manifested in each tuple in the resulting set `ultimateRaces`.

For our running example, recall that at the end of the first iteration, the set `ultimateRacesWithObject` contains the lone tuple T3 along with abstract object `[hi]`, that is, local variables `v2` and `v5` are deemed to point to that thread-escaping abstract object just before program points `p1` and `p2`, respectively (see heap abstraction (a) in Figure 4.3). Since `[hi]` satisfies the above two checks, we have $H = \{\text{hi}\}$ and $A = \{\text{hi}\}$, and the refined Σ uses a k value of two for site `hi` and one for all other sites. Since H is non-empty, our algorithm iterates using the refined Σ .

	# classes	# bytecodes	brief description
<code>elevator</code>	1066	109831	Discrete event simulator
<code>tsp</code>	1068	110582	TSP solver from ETH
<code>hedc</code>	1592	278010	Web crawler from ETH
<code>ftp</code>	1905	348813	Apache FTP server
<code>pool</code>	1121	122187	Apache pooling library
<code>jdbf</code>	1739	291392	Object-relational mapping system
<code>jtds</code>	1801	301231	JDBC driver
<code>derby</code>	3428	721912	Apache relational database engine

Table 4.1: Benchmarks.

At the end of the second iteration, `ultimateRacesWithObject` still contains tuple T3, but this time with abstract object `[hi::hc]`, that is, `v2` and `v5` are deemed to point to that thread-escaping abstract object just before `p1` and `p2`, respectively (see heap abstraction (d) in Figure 4.3). Since `[hi::hc]` satisfies the above two checks, we have $H = \{\text{hi}, \text{hc}\}$ and $A = \{[\text{hi}], [\text{hi}::\text{hc}]\}$, and the refined Σ uses a k value of three for site `hi`, two for site `hc`, and one for all other sites. Once again, since H is non-empty, our algorithm iterates using the refined Σ .

At the end the third iteration, `ultimateRacesWithObject` is empty since `v2` is deemed to point to thread-escaping abstract object `[hi::hc::ht]` just before `p1` in abstract context `[ht]`, whereas `v5` is deemed to point to a different thread-escaping abstract object `[hi::hc]` just before `p2` in abstract context `[ht]` (see heap abstraction (c) in Figure 4.3). Thus, H is empty and our algorithm terminates without reporting a false race.

4.6 Experiments

We have implemented our demand-driven race detection algorithm and applied it to a suite of eight multithreaded Java programs. Table 4.1 provides a brief description of each program along with the number of classes and the number of bytecodes in the program. The numbers correspond to code that is deemed reachable from the main method of each program in a context insensitive call graph that is computed by

	running time				# iters. in d.d.	avg. time per iter. in d.d.
	k=1	k=2	k=3	d.d.		
elevator	3m21s	3m45s	12m14s	8m24s	5	1m41s
tsp	3m37s	4m03s	10m41s	5m00s	2	2m30s
hedc	10m04s	9m31s	38m12s	23m17s	7	3m20s
ftp	15m15s	13m51s	54m15s	30m23s	8	3m48s
pool	4m48s	3m40s	9m43s	7m59s	5	1m36s
jdbf	8m40s	8m52s	29m31s	20m52s	7	2m59s
jtds	11m33s	14m28s	37m12s	31m19s	7	4m28s
derby	34m31s	?	?	132m	8	16m30s

Table 4.2: Experimental results: Comparison of running time.

Spark [52], a 0-CFA-based may alias analysis with on-the-fly call graph construction provided in the Soot compiler framework [81]. The experiments were performed on a 2.4GHz machine with 4GB memory.

Table 4.2 presents the total running time of our previous algorithm using $k = 1$, $k = 2$, and $k = 3$, and our demand-driven algorithm, denoted d.d. Notice that our previous algorithm using $k = 2$ and $k = 3$ runs out of memory for the largest program **derby**. The “# iters. in d.d.” column denotes the total number of times our demand-driven algorithm iterates for each program. It ranges from two for **tsp** to eight for **ftp** and **derby**. The “avg. time per iter. in d.d.” column denotes the average time per iteration, computed by dividing the total running time by the total number of iterations. Notice that it is less than the total running time of our previous algorithm using $k = 1$ for each program. This is because the results computed in the first iteration of our demand-driven algorithm (which is equivalent to our previous algorithm using $k = 1$) such as those computed using Soot are reused in subsequent iterations and, more importantly, each subsequent iteration increases the k value for only a tiny fraction of the object allocation sites in each program. This is evident from the numbers in Table 4.3 which presents for each program the profile of function Σ after our demand-driven algorithm terminates, namely, each column labeled $n = 1, 2, \dots, 8$ provides the size of the set $\{ h \mid \Sigma(h) = n \}$. Note that our demand-driven algorithm uses a k value greater than one for less than 0.01% of all

	k value							
	1	2	3	4	5	6	7	8
elevator	3152	0	1	4	2	-	-	-
tsp	3246	1	-	-	-	-	-	-
hedc	8720	32	16	11	13	10	8	-
ftp	10976	21	18	15	11	15	14	11
pool	3079	2	1	6	2	-	-	-
jdbf	9131	27	19	10	12	9	5	-
jtds	9507	39	14	7	5	3	1	-
derby	21505	97	34	21	9	11	8	5

Table 4.3: Experimental results: Number of sites in Σ after last iteration.

sites in each program.

To compare scalability, Table 4.4 presents the size of $\mathcal{A}$ used by each of the three variants of our previous algorithm as well as that used in each iteration of our demand-driven algorithm. Recall from Section 4.4 that $\mathcal{A}$ is a close over-approximation of both sets $\mathbb{C}$ and $\mathbb{O}$ of abstract contexts and abstract objects, respectively, that is pre-computed before performing k -object-sensitive analysis. The exponential growth in the size of $\mathcal{A}$ is evident in our previous algorithm as the user-supplied k value is increased from one to three. In contrast, the size of $\mathcal{A}$ even after the last iteration of our demand-driven algorithm for each program is much smaller than the size of $\mathcal{A}$ for our previous algorithm using $k = 2$.

To compare precision, Table 4.5 presents the size of set $\pi_{3,6}(\text{ultimateRaces})$, that is, the number of ultimate race pairs reported by the three variants of our previous algorithm as well as that computed after each iteration of our demand-driven algorithm (though only that computed after the last iteration is reported for manual inspection). The “false” column presents the number of false races reported by our previous algorithm using $k = 3$ and our demand-driven algorithm. An exception is **derby** for which we present the number of false races reported by our previous algorithm using $k = 1$ since it runs out of memory using $k = 3$. The numbers of real races reported by both algorithms are the same since both are sound (under the assumption that the programs are complete, that is, they do not have missing

	k=1	k=2	k=3
elevator	3160	8801	133961
tsp	3248	9026	134500
hedc	8811	28019	273201
ftp	11082	36041	364748
pool	3091	8726	133885
jdbf	9214	31312	296017
jtds	9577	34210	310321
derby	21691	?	?

	in d.d. after iteration #							
	1	2	3	4	5	6	7	8
elevator	3160	3160	3160	3160	3160	-	-	-
tsp	3248	3248	-	-	-	-	-	-
hedc	8811	9439	9847	11181	11982	11993	11993	-
ftp	11082	11983	12382	14060	14885	14901	14907	14907
pool	3091	3200	3200	3200	3200	-	-	-
jdbf	9214	10292	10994	11376	11412	11415	11415	-
jtds	9577	11321	11430	11526	11592	11592	11592	-
derby	21691	24102	25124	25441	25523	25523	25523	25523

Table 4.4: Experimental results: Comparison of numbers of abstract contexts.

	k=1	k=2	k=3	false
elevator	1	1	1	1
tsp	19	19	19	8
hedc	864	257	215	56
ftp	1105	451	378	107
pool	141	31	31	2
jdbf	930	381	302	20
jtds	1003	423	322	34
derby	1445	?	?	427

	in d.d. after iteration #								false
	1	2	3	4	5	6	7	8	
elevator	1	1	1	1	1	-	-	-	1
tsp	19	19	-	-	-	-	-	-	8
hedc	864	334	276	228	207	207	207	-	48
ftp	1105	525	431	382	368	368	368	368	97
pool	141	37	37	37	37	-	-	-	8
jdbf	930	441	397	335	311	311	311	-	29
jtds	1003	482	402	356	342	322	322		34
derby	1445	1218	1176	1121	1096	1096	1096	1096	28

Table 4.5: Experimental results: Comparison of numbers of ultimate race pairs.

callers or callees and they do not use dynamic class loading and reflection). For `pool` and `jdbf`, our demand-driven algorithm reports more false positives than our previous algorithm using $k = 3$, illustrating the non-optimality of our demand-driven algorithm, but for the remaining six programs, it reports an equal or fewer number of false positives.

Finally, notice that the number of pairs of ultimate races remains constant in later iterations of our demand-driven algorithm for each program although the k values for certain allocation sites are increased in those iterations. This is because tuples manifesting races on fields of objects allocated at these sites continue to be retained in set `ultimateRaces` either because they are real races (and will never be eliminated) or they are false positives arising because of a different source of imprecision than small k values of the sites we increment. The computation performed by these iterations, however, does not go in vain: suppose a race is deemed to occur on a field f of an

object abstracted by $[h_n :: \dots :: h_1]$ in iteration i of our algorithm. Then, increasing the k value of each of $h_1, \dots, h_n$ is useful even if the race is not eliminated in iteration $(i + 1)$, because in that iteration, we have more information to report to the user, in particular, we can now report that the abstract object on whose field f the race is deemed to occur is $[h_n :: \dots :: h_1 :: h_0]$. This is useful regardless of whether the reported race turns out to be a real race or a false positive, because sites $h_1, \dots, h_n$ are typically increasingly deeply nested sites occurring in library code and the race itself occurs between accesses in library code, and providing a “shallower” site like h_0 occurring in application code helps the user comprehend how the (presumably more familiar) application code causes the race in the (presumably less familiar) library code. This also explains why the size of $\mathcal{A}$ is constant in later iterations although the k values for certain allocation sites are increased in those iterations: abstract object $[h_n :: \dots :: h_1]$ in $\mathcal{A}$ in iteration i gets replaced by $[h_n :: \dots :: h_1 :: h_0]$ in iteration $(i + 1)$.

4.7 Related Work

We have already discussed related work on race detection in Chapter 2. In this section, we survey related work on demand-driven alias analysis and its applications. We have already mentioned that Milanova et al. [60, 61] present several dimensions in which their k -object-sensitive analysis is parameterized. A key challenge lies in choosing a form of parameterization and instantiating it in a manner that is most effective for a client of the analysis. Our race detection client only exploits the form of parameterization which allows different k values for different object allocation sites.

Our iterative demand-driven approach is inspired by that of Plevyak and Chien [67] and Guyer and Lin [40, 41]. Plevyak and Chien present an iterative demand-driven approach to strike a trade-off between scalability and precision in the context of type inference for a dynamically-typed object-oriented language. The approach attempts to resolve type conflicts in successive iterations by performing *function splitting* which provides context sensitivity or *container splitting* which divides object allocation sites so that a single site can generate objects of different types. It inserts run time type checks for conflicts that cannot be eliminated. A key difference

between our approach and theirs is that we do not need to distinguish between the two kinds of splitting since we use k -object-sensitive analysis which treats abstract contexts and abstract objects uniformly: increasing the k values for object allocation sites increases both the number of abstract contexts (i.e., performs function splitting) and the number of abstract objects (i.e., performs container splitting).

Guyer and Lin [40, 41] present a client-driven alias analysis for C and its effectiveness on five temporal safety property checking clients. The approach begins with a flow and context insensitive alias analysis. It consists of a *monitor* that builds a dependence graph during the analysis to keep track of the effects of imprecision that occur during the analysis due to flow and context insensitivity. A client of the analysis reports back to an *adaptor* the set of memory locations for which it requires more precision, which for our race detection client may be viewed as the allocation sites of objects on whose fields races deemed to occur. The adaptor views these memory locations as *symptoms* of imprecision and uses the dependence graph built by the monitor to determine their *causes*, in particular it determines which assignments to treat flow sensitively and which methods to treat context sensitively in the subsequent iteration. A key difference between our approach and theirs is that our approach lacks a monitor and treats each symptom itself as a cause of imprecision. This is indeed mostly the case for us in practice, but as illustrated for the `pool` and `jdbf` benchmarks in Section 4.6, it can cause our approach to report more false positives than a non-demand-driven approach. Another key difference is that our approach cannot eliminate imprecision arising due to flow insensitivity though the need for it seems more critical to their clients which check temporal safety properties for C.

Heintz and Tardieu [44] present a demand-driven alias analysis for C that performs provably optimal computation to determine the points-to sets for variables provided by a client. They show how to apply their analysis to a call graph construction client for C in the presence of function pointers. The key difference between this approach and the ones discussed above is that it computes a partial solution sufficient to answer the client's queries whereas the rest compute an exhaustive solution with varying precision for different parts of the program.

Sridharan et al. [77, 78] present a refinement-based alias analysis for Java and

apply it to a type cast checking client. Their approach computes a partial solution sufficient to determine the points-to sets for variables provided by a client but it also successively refines parts of the solution to precisely handle those method calls/returns and field reads/writes that are key to precisely answering the client's queries.

Zheng and Rugina [93] present a demand-driven alias analysis for C that also computes a partial solution sufficient to answer may alias queries posed by clients. Unlike most approaches, however, their approach does not require computing and intersecting points-to sets for answering such queries.

Chapter 5

Conclusion

We set out to develop a static race detection tool for Java with five criteria we felt such a tool must address to be effective:

1. *Precision.* Does the tool have a tolerable false positive rate on real-world Java programs?
2. *Soundness.* Does the tool detect all races, modulo standard unsoundness assumptions made by static analyses for Java?
3. *Scalability.* Is the tool fully automatic and capable of checking large programs?
4. *Open Programs.* Does the tool handle open programs such as libraries?
5. *Counterexamples.* Does the tool provide sufficient information to identify and fix the bugs manifested in its race reports?

The techniques proposed in this thesis have reasonably addressed all of the above five criteria. Chapter 2 presented a novel static race detection algorithm that focused on precision and described how to handle open programs and generate counterexamples. Chapter 3 presented a novel technique for soundly correlating locks with the memory locations they guard, called conditional must not alias analysis, and used it to replace the unsound lockset analysis employed in the original race detection algorithm. Lastly, Chapter 4 presented a novel demand-driven approach,

which improved the scalability of the race detection algorithm by enabling an adaptive k -object-sensitive analysis capable of using different k values for different object allocation sites in the given program.

We implemented the resulting algorithm in a tool **Chord** and applied it to a suite of eight multithreaded Java programs. Our approach found tens to hundreds of previously unknown concurrency bugs in mature and widely used programs in our benchmark suite. The usefulness of our approach was attested by the fact that many of these bugs were fixed by the programs' developers upon reporting.

5.1 Future Work

The work presented in this thesis can be extended in several ways. First, developers of our benchmark programs were often reluctant to fix the races we reported due to concerns of introducing deadlocks. Since races do not have fail-stop behavior unlike deadlocks, however, they eventually chose to fix the races, but this experience convinced us that static deadlock detection is an important open problem. Furthermore, the static analyses required for proving deadlock freedom for Java bear many similarities to those for proving race freedom, for instance, a precise may alias analysis is required to distinguish statically between different locks and a precise form of must alias analysis is required to statically identify reentrant locks. Imprecision in either analysis leads to false lock cycles such as those reported by existing state-of-the-art static deadlock detectors for Java [91].

Besides race freedom and deadlock freedom, another desirable lightweight concurrency correctness property that has gained significant attention in recent years is atomicity [35, 36] and atomic sets [82]. Race detection is a first step in many atomicity checkers, in particular, atomicity checkers based on Lipton's theory of reduction [54] must show that each statement accessing a memory location is both a *left mover* and a *right mover*, which is done by proving the absence of races on that location. Existing static atomicity checkers, however, are limited by the effectiveness of the underlying static race detector. We hope to apply our static race detection algorithm to perform effective static atomicity checking.

Another line of work we intend to pursue is exploiting the global information about locks that is at the disposal of our static race detection algorithm to:

1. automatically remove unnecessary synchronization [6, 9, 10, 16, 72, 89], which in turn would improve performance and address the other most significant complaint developers raised while fixing the races we reported,
2. automatically suggest fixes to incorrect synchronization in legacy programs [33], which would alleviate the manual burden of inspecting and fixing races, and
3. automatically introduce necessary synchronization in future programs [25, 57].

Solving each of these problems would require taking into account performance metrics as well as correctness metrics like race freedom, deadlock freedom, and atomicity. To the best of our knowledge, all existing work addresses only a subset of the above metrics. We believe it would be more useful, however, to address all these concerns simultaneously, if possible integrating results on static race detection, static deadlock detection, and static atomicity checking.

Finally, our current static race detection algorithm does not handle open programs such as libraries soundly. Even specifying a concurrency property like race freedom, deadlock freedom, atomicity, and, more generally, “thread safety”, for an open program is challenging. It would be useful to design formal specifications for these problems and devise sound static analyses for solving them.

Appendix A

Proof of Type Preservation

We state a useful fact of environment abstraction (Figure 3.7 (c)) that is needed in proving type preservation.

Fact A.1. (Weakening of loop set) *If $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ and $W' \subseteq W$ then $W' \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$.*

Lemma A.2. (Type Preservation) *If $s, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C$ and $W, \Pi, \Gamma \vdash s : \Gamma', K$ and $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ then $W \vdash (\pi', \rho') \preceq (\Pi, \Gamma')$ and $C \preceq K$.*

Proof. By induction on the structure of the derivation of $s, W, \pi, \rho, \sigma \Downarrow \pi', \rho', \sigma', C$. There are ten cases depending on which one of rules (3.1)–(3.10) in Figure 3.5 is used last in the derivation. For brevity, we only provide the proof for the two most interesting cases.

1. Rule (3.5). We have $s \equiv v_1.f = v_2$. There are two sub-cases depending upon whether $\rho(v_2)$ is null or non-null. We only prove the latter more interesting sub-case. We have:

- (a) $\rho(v_1) = o_1$ and $\rho(v_2) = o_2$

- (b) $s, W, \pi, \rho, \sigma \Downarrow \pi, \rho, \sigma[(o_1, f) \mapsto o_2], \{o_1 \triangleright o_2\}$

From $s \equiv v_1.f = v_2$ and hypothesis $W, \Pi, \Gamma \vdash s : \Gamma', K$ of the lemma and rule (3.15) in Figure 3.8, we have:

- (c)
$$K = \begin{cases} \{\tau_1 \supseteq \tau_2\} & \text{if } \Gamma(v_1) = \tau_1 \text{ and } \Gamma(v_2) = \tau_2 \\ \emptyset & \text{otherwise} \end{cases}$$

To prove:

$$(I) W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$$

$$(II) \{o_1 \triangleright o_2\} \preceq K$$

We have (I) immediately from hypothesis $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ of the lemma.

We will next prove that $\exists \tau'_1, \tau'_2$:

$$(III) (\tau'_1 \triangleright \tau'_2) \in K$$

$$(IV) (o_1, o_2) \propto (\tau'_1, \tau'_2)$$

From (III), (IV), and the definition of $\preceq$ in Figure 3.7 (b), we will have (II).

Proof of (III) and (IV):

From hypothesis $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ of the lemma and item (2) and item (3)(b) in Figure 3.7 (c), we have:

$$(d) \forall v \in \mathbb{V} : \rho(v) \preceq \Gamma(v)$$

$$(e) \forall w \in \mathbb{W} : \exists n \in \mathbb{N} : \forall v \in \mathbb{V} : ((\rho(v) = o \wedge \Gamma(v) = \tau \wedge \tau.\mathbf{\Pi}(w) = 1) \Rightarrow o.\mathbf{\pi}(w) = n)$$

From (a), (d), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(f) \Gamma(v_1) = \tau_1 \text{ and } \Gamma(v_2) = \tau_2$$

$$(g) o_1 \preceq \tau_1 \text{ and } o_2 \preceq \tau_2$$

From (c) and (f), we have:

$$(h) K = \{\tau_1 \triangleright \tau_2\}$$

From (e), (a), and (f), we have:

$$(i) \forall w \in \mathbb{W} : \exists n \in \mathbb{N} : ((\tau_1.\mathbf{\Pi}(w) = 1 \Rightarrow o_1.\mathbf{\pi}(w) = n) \wedge (\tau_2.\mathbf{\Pi}(w) = 1 \Rightarrow o_2.\mathbf{\pi}(w) = n))$$

Choose $\tau'_1 = \tau_1$ and $\tau'_2 = \tau_2$. Then, we have (III) from (h) and we have (IV) from (g), (i), and the definition of $\propto$ in Figure 3.7 (b).

2. Rule (3.10). We have $s \equiv \text{while}^w (*) \text{ do } s'$ and:

$$(a) s', W \cup \{w\}, \pi[w \mapsto \pi(w) + 1], \rho, \sigma \Downarrow \pi', \rho', \sigma', C_1$$

$$(b) s, W, \pi', \rho', \sigma' \Downarrow \pi'', \rho'', \sigma'', C_2$$

From $s \equiv \text{while}^w (*) \text{ do } s'$ and hypothesis $W, \Pi, \Gamma \vdash s : \Gamma', K$ of the lemma and

rule (3.18) in Figure 3.8, we have:

- (c) $W, \Pi, \Gamma \vdash s : \Gamma, K$
- (d) $W \cup \{w\}, \Pi, \Gamma^{w+} \vdash s' : \Gamma, K$
- (e) $\Pi(w) \neq 0$

To prove:

- (I) $W \vdash (\pi'', \rho'') \preceq (\Pi, \Gamma)$
- (II) $(C_1 \cup C_2) \preceq K$

From hypothesis $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ of the lemma and Figure 3.7 (c), we have:

- (f) $\forall w' \in W : \pi(w') \preceq \Pi(w')$
- (g) $\forall v \in \mathbb{V} : \rho(v) \preceq \Gamma(v)$
- (h) $\forall w' \in \mathbb{W} : \exists n' \in \mathbb{N} :$

$$(\Pi(w') = 1 \Rightarrow \pi(w') = n') \wedge$$

$$(\forall v \in \mathbb{V} : ((\rho(v) = o \wedge \Gamma(v) = \tau \wedge \tau.\Pi(w') = 1) \Rightarrow o.\pi(w') = n'))$$

We will next prove:

- (III) $W \cup \{w\} \vdash (\pi[w \mapsto \pi(w) + 1], \rho) \preceq (\Pi, \Gamma^{w+})$

Then, from (a), (d), (III), and the induction hypothesis, we will have:

- (i) $W \cup \{w\} \vdash (\pi', \rho') \preceq (\Pi, \Gamma)$
- (j) $C_1 \preceq K$

From (i) and Fact A.1, we will have:

- (k) $W \vdash (\pi', \rho') \preceq (\Pi, \Gamma)$

From (b), (c), (k), and the induction hypothesis, we will have:

- (l) $W \vdash (\pi'', \rho'') \preceq (\Pi, \Gamma)$
- (m) $C_2 \preceq K$

From (l), we will have (I). From (j), (m), and the definition of $\preceq$ in Figure 3.7 (b), we will have (II).

Proof of (III):

From (e), we have:

- (n) $\Pi(w) = 1 \vee \Pi(w) = \top$

From $(\pi(w) + 1) > 0$, (n), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(o) \ (\pi(w) + 1) \preceq \Pi(w)$$

From (f) and (o), we have:

$$(p) \ \forall w' \in W \cup \{w\} : \pi[w \mapsto \pi(w) + 1](w') \preceq \Pi(w')$$

From (g), Definition 3.3, and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(q) \ \forall v \in \mathbb{V} : \rho(v) \preceq \Gamma^{w+}(v)$$

We will next prove:

$$(IV) \ \exists n \in \mathbb{N} :$$

$$\begin{aligned} & (\Pi(w) = 1 \Rightarrow (\pi[w \mapsto \pi(w) + 1])(w) = n) \wedge \\ & (\forall v \in \mathbb{V} : ((\rho(v) = o \wedge \Gamma^{w+}(v) = \tau \wedge \tau.\mathbf{\Pi}(w) = 1) \Rightarrow o.\mathbf{\pi}(w) = n)) \end{aligned}$$

From (h) and (IV), we will have:

$$(r) \ \forall w' \in \mathbb{W} : \exists n' \in \mathbb{N} :$$

$$\begin{aligned} & (\Pi(w') = 1 \Rightarrow (\pi[w \mapsto \pi(w) + 1])(w') = n') \wedge \\ & (\forall v \in \mathbb{V} : ((\rho(v) = o \wedge \Gamma^{w+}(v) = \tau \wedge \tau.\mathbf{\Pi}(w') = 1) \Rightarrow o.\mathbf{\pi}(w') = n')) \end{aligned}$$

From (p), (q), (r), and Figure 3.7 (c), we will have (III).

Proof of (IV):

Choose $n = (\pi[w \mapsto \pi(w) + 1])(w)$, whence we trivially have:

$$(s) \ \Pi(w) = 1 \Rightarrow (\pi[w \mapsto \pi(w) + 1])(w) = n$$

From Definition 3.3, we have $\forall v \in \mathbb{V} : (\Gamma^{w+}(v) = \tau \Rightarrow \tau.\mathbf{\Pi}(w) \neq 1)$ whence we trivially have:

$$(t) \ \forall v \in \mathbb{V} : ((\rho(v) = o \wedge \Gamma^{w+}(v) = \tau \wedge \tau.\mathbf{\Pi}(w) = 1) \Rightarrow o.\mathbf{\pi}(w) = n)$$

From (s) and (t), we have (IV).

□

Lemma A.3. *If $C \preceq K$ then $\forall n \geq 1 : C^n \preceq K^n$.*

Proof. By induction on n .

Base case ($n = 1$): Suppose $C \preceq K$. To prove $C^1 \preceq K^1$. From item (1) of Definition 3.7, we have $C^1 = C$. Likewise, from item (1) of Definition 3.10, we have $K^1 = K$. From $C \preceq K$ and $C^1 = C$ and $K^1 = K$, we have $C^1 \preceq K^1$.

Inductive step: Suppose:

$$(a) \ C \preceq K$$

To prove $C^{n+1} \preceq K^{n+1}$. Consider any o_1 and o_3 such that:

$$(b) \ (o_1 \triangleright o_3) \in C^{n+1}$$

Then, from the definition of $\preceq$ in Figure 3.7 (b), it suffices to prove that $\exists \tau_1, \tau_5$:

$$(I) \ (\tau_1 \triangleright \tau_5) \in K^{n+1}$$

$$(II) \ (o_1, o_3) \propto (\tau_1, \tau_5)$$

From (b) and item (2) of Definition 3.7, we have $\exists o_2$:

$$(c) \ (o_1 \triangleright o_2) \in C^n$$

$$(d) \ (o_2 \triangleright o_3) \in C$$

From the induction hypothesis and (a), we have:

$$(e) \ C^n \preceq K^n$$

From (c), (e), and the definition of $\preceq$ in Figure 3.7 (b), we have $\exists \tau_1, \tau_2$:

$$(f) \ (\tau_1 \triangleright \tau_2) \in K^n$$

$$(g) \ (o_1, o_2) \propto (\tau_1, \tau_2)$$

From (d), (a), and the definition of $\preceq$ in Figure 3.7 (b), we have $\exists \tau_3, \tau_4$:

$$(h) \ (\tau_3 \triangleright \tau_4) \in K$$

$$(i) \ (o_2, o_3) \propto (\tau_3, \tau_4)$$

From (g) and item (2) in Figure 3.7 (b), we have:

$$(j) \ o_2 \preceq \tau_2$$

From (i) and item (1) in Figure 3.7 (b), we have:

$$(k) \ o_2 \preceq \tau_3$$

From (j) and the definitions in Figure 3.7 (a), we have:

$$(l) \ (o_2.\hat{\mathbf{h}} = \tau_2.\hat{\mathbf{h}} \vee \tau_2.\hat{\mathbf{h}} = \top) \wedge \forall w \in \mathbb{W} : ((o_2.\boldsymbol{\pi}(w) = 0 \wedge \tau_2.\boldsymbol{\Pi}(w) = 0) \vee (o_2.\boldsymbol{\pi}(w) > 0 \wedge \tau_2.\boldsymbol{\Pi}(w) = 1) \vee \tau_2.\boldsymbol{\Pi}(w) = \top)$$

From (k) and the definitions in Figure 3.7 (a), we have:

$$(m) (o_2.\hat{\mathbf{h}} = \tau_3.\hat{\mathbf{h}} \vee \tau_3.\hat{\mathbf{h}} = \top) \wedge \forall w \in \mathbb{W} : ((o_2.\boldsymbol{\pi}(w) = 0 \wedge \tau_3.\boldsymbol{\Pi}(w) = 0) \vee (o_2.\boldsymbol{\pi}(w) > 0 \wedge \tau_3.\boldsymbol{\Pi}(w) = 1) \vee \tau_3.\boldsymbol{\Pi}(w) = \top)$$

From (l) and (m), we have:

$$(n) (\tau_2.\hat{\mathbf{h}} = \tau_3.\hat{\mathbf{h}} \vee \tau_2.\hat{\mathbf{h}} = \top \vee \tau_3.\hat{\mathbf{h}} = \top) \wedge \forall w \in \mathbb{W} : (\tau_2.\boldsymbol{\Pi}(w) = \tau_3.\boldsymbol{\Pi}(w) \vee \tau_2.\boldsymbol{\Pi}(w) = \top \vee \tau_3.\boldsymbol{\Pi}(w) = \top)$$

From (n) and Definition 3.8, we have:

$$(o) \tau_2 \sim \tau_3$$

From (f), (h), (o), and item (2) of Definition 3.10, we have $\exists \tau_5$:

$$(p) (\tau_1 \supseteq \tau_5) \in K^{n+1}$$

(q) τ_5 satisfies conditions (a) and (b) in item (2) of Definition 3.10

From (p), we have (I).

From (g) and item (1) in Figure 3.7 (b), we have:

$$(r) o_1 \preceq \tau_1$$

From (h) and item (2) in Figure 3.7 (b), we have:

$$(s) o_3 \preceq \tau_4$$

From (s), (q), and the definitions in Figure 3.7 (a), we have:

$$(t) o_3 \preceq \tau_5$$

We will next prove:

$$(III) \forall w \in \mathbb{W} : ((\tau_1.\boldsymbol{\Pi}(w) = 1 \wedge \tau_5.\boldsymbol{\Pi}(w) = 1) \Rightarrow o_1.\boldsymbol{\pi}(w) = o_3.\boldsymbol{\pi}(w))$$

From (r), (t), (III), and the definition of $\propto$ in Figure 3.7 (b), we will have (II). Consider any $w \in \mathbb{W}$ and suppose:

$$(u) \tau_1.\boldsymbol{\Pi}(w) = 1 \wedge \tau_5.\boldsymbol{\Pi}(w) = 1$$

To prove (III), it suffices to prove:

$$(IV) o_1.\boldsymbol{\pi}(w) = o_3.\boldsymbol{\pi}(w)$$

From (u), we have $\tau_5.\boldsymbol{\Pi}(w) = 1$, which combined with (q) yields (see condition (b) in item (2) of Definition 3.10):

$$(v) \tau_4.\boldsymbol{\Pi}(w) = 1$$

$$(w) \tau_1.\mathbf{\Pi}(w) \neq 1 \vee \tau_5.\mathbf{\Pi}(w) \neq 1 \vee \tau_2.\mathbf{\Pi}(w) = \tau_3.\mathbf{\Pi}(w) = 1$$

From (u) and (w), we have:

$$(x) \tau_2.\mathbf{\Pi}(w) = \tau_3.\mathbf{\Pi}(w) = 1$$

From (g), $\tau_1.\mathbf{\Pi}(w) = 1$ obtained from (u), $\tau_2.\mathbf{\Pi}(w) = 1$ obtained from (x), and item (3) in Figure 3.7 (b), we have:

$$(y) o_1.\boldsymbol{\pi}(w) = o_2.\boldsymbol{\pi}(w)$$

From (i), $\tau_3.\mathbf{\Pi}(w) = 1$ obtained from (x), $\tau_4.\mathbf{\Pi}(w) = 1$ obtained from (v), and item (3) in Figure 3.7 (b), we have:

$$(z) o_2.\boldsymbol{\pi}(w) = o_3.\boldsymbol{\pi}(w)$$

From (y) and (z), we have (IV). □

Lemma A.4. *If $C \preceq K$ and $h \in DR_K(H)$ then $h \in DR_C(H)$.*

Proof. Suppose:

$$(a) C \preceq K$$

$$(b) h \in DR_K(H)$$

To prove $h \in DR_C(H)$. Consider any o_1, o_2 , and o such that:

$$(c) (o_1 \triangleright o) \in C^+$$

$$(d) (o_2 \triangleright o) \in C^+$$

$$(e) o_1.\mathbf{h} \in H$$

$$(f) o_2.\mathbf{h} \in H$$

$$(g) o.\mathbf{h} = h$$

Then, from the first equation in Figure 3.9, it suffices to prove: (I) $o_1 = o_2$, that is:

From (a) and Lemma A.3, we have:

$$(h) \forall n \geq 1 : C^n \leq K^n$$

From (c) and definition 3.7, we have:

$$(i) \exists n_1 \geq 1 : (o_1 \triangleright o) \in C^{n_1}$$

From (d) and definition 3.7, we have:

$$(j) \exists n_2 \geq 1 : (o_2 \triangleright o) \in C^{n_2}$$

From (h), (i), and the definition of $\preceq$ in Figure 3.7 (b), we have $\exists \tau_1, \tau_3$:

$$(k) (\tau_1 \supseteq \tau_3) \in K^{n_1}$$

$$(l) (o_1, o) \propto (\tau_1, \tau_3)$$

From (h), (j), and the definition of $\preceq$ in Figure 3.7 (b), we have $\exists \tau_2, \tau_4$:

$$(m) (\tau_2 \supseteq \tau_4) \in K^{n_2}$$

$$(n) (o_2, o) \propto (\tau_2, \tau_4)$$

From (l) and the definition of $\propto$ in Figure 3.7 (b), we have:

$$(o) o_1 \preceq \tau_1$$

$$(p) o \preceq \tau_3$$

$$(q) \forall w \in \mathbb{W} : ((\tau_1.\mathbf{\Pi}(w) = 1 \wedge \tau_3.\mathbf{\Pi}(w) = 1) \Rightarrow o_1.\mathbf{\pi}(w) = o.\mathbf{\pi}(w))$$

From (n) and the definition of $\propto$ in Figure 3.7 (b), we have:

$$(r) o_2 \preceq \tau_2$$

$$(s) o \preceq \tau_4$$

$$(t) \forall w \in \mathbb{W} : ((\tau_2.\mathbf{\Pi}(w) = 1 \wedge \tau_4.\mathbf{\Pi}(w) = 1) \Rightarrow o_2.\mathbf{\pi}(w) = o.\mathbf{\pi}(w))$$

From (o), (e), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(u) \tau_1.\hat{\mathbf{h}} \in H \cup \{\top\}$$

From (r), (f), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(v) \tau_2.\hat{\mathbf{h}} \in H \cup \{\top\}$$

From (p), (s), the definition of $\preceq$ in Figure 3.7 (a), and Definition 3.8, we have:

$$(w) \tau_3 \sim \tau_4$$

From (p), (g), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(x) \tau_3.\hat{\mathbf{h}} \in \{h, \top\}$$

From (s), (g), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(y) \tau_4.\hat{\mathbf{h}} \in \{h, \top\}$$

From (k), (m), (u), (v), (w), (x), (y), and the second equation in Figure 3.9, we have:

$$(z) \tau_1 = \tau_2$$

$$(a') \tau_1 < \top$$

$$(b') \forall w \in \mathbb{W} : (\tau_1.\mathbf{\Pi}(w) = 1 \Rightarrow \tau_3.\mathbf{\Pi}(w) = \tau_4.\mathbf{\Pi}(w) = 1)$$

From (z), (a'), and Definition 3.9, we have:

$$(c') \tau_1.\hat{\mathbf{h}} = \tau_2.\hat{\mathbf{h}} \neq \top$$

$$(d') \forall w \in \mathbb{W} : (\tau_1.\mathbf{\Pi}(w) = \tau_2.\mathbf{\Pi}(w) = 0 \vee \tau_1.\mathbf{\Pi}(w) = \tau_2.\mathbf{\Pi}(w) = 1)$$

From (o), (r), (c'), and the definition of $\preceq$ in Figure 3.7 (a), we have:

$$(e') o_1.\mathbf{h} = o_2.\mathbf{h}$$

$$(f') \forall w \in \mathbb{W} : (\tau_1.\mathbf{\Pi}(w) = \tau_2.\mathbf{\Pi}(w) = 0 \Rightarrow o_1.\boldsymbol{\pi}(w) = o_2.\boldsymbol{\pi}(w))$$

From (q), (t), and (b'), we have:

$$(g') \forall w \in \mathbb{W} : (\tau_1.\mathbf{\Pi}(w) = \tau_2.\mathbf{\Pi}(w) = 1 \Rightarrow o_1.\boldsymbol{\pi}(w) = o_2.\boldsymbol{\pi}(w))$$

From (d'), (f'), and (g'), we have:

$$(h') \forall w \in \mathbb{W} : o_1.\boldsymbol{\pi}(w) = o_2.\boldsymbol{\pi}(w)$$

From (e') and (h'), we have (I). □

Bibliography

- [1] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA'91)*, pages 234–243, 1991.
- [2] R. Agarwal, A. Sasturkar, Wang L, and S. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 233–242, 2005.
- [3] R. Agarwal and S. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, pages 149–160, 2004.
- [4] S. Agarwal, R. Barik, V. Sarkar, and R. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pages 183–193, 2007.
- [5] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 342–354, 1998.
- [6] J. Aldrich, E. Sirer, C. Chambers, and S. Eggers. Comprehensive synchronization elimination for Java. *Science of Computer Programming*, 47(2-3):91–120, 2003.
- [7] D. Bacon, R. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented*

- Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 382–400, 2000.
- [8] M. Berndt, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 103–114, 2003.
- [9] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 20–34, 1999.
- [10] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 35–46, 1999.
- [11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 211–230, 2002.
- [12] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pages 56–69, 2001.
- [13] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [14] G. Cheng, M. Feng, C. Leiserson, K. Randall, and A. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, 1998.
- [15] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented*

- Programming, Systems, Languages, and Applications (OOPSLA '99)*, pages 1–19, 1999.
- [16] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, 2003.
- [17] J. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’02)*, pages 258–269, 2002.
- [18] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, 2001.
- [19] J. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.
- [20] M. Christiaens and K. Brooschere. TRaDe: A topological approach to on-the-fly race detection in Java programs. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM’01)*, pages 105–116, 2001.
- [21] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS’77)*, pages 77–94, 1977.
- [22] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP’90)*, pages 1–10, 1990.
- [23] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging (PADD’91)*, pages 85–96, 1991.

- [24] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, pages 12–22, 2004.
- [25] M. Emmi, J. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, 2007.
- [26] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 237–252, 2003.
- [27] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software (SPIN'04)*, pages 252–266, 2004.
- [28] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming (ESOP'99)*, pages 91–108, 1999.
- [29] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–232, 2000.
- [30] C. Flanagan and S. Freund. Detecting race conditions in large programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 90–96, 2001.
- [31] C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multi-threaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04)*, pages 256–267, 2004.
- [32] C. Flanagan and S. Freund. Type inference against races. In *Proceedings of the 11th International Static Analysis Symposium (SAS'04)*, pages 116–132, 2004.

- [33] C. Flanagan and S. Freund. Automatic synchronization correction. In *Workshop on Synchronization and Concurrency in Object-Oriented Language (SCOOL'05)*, 2005.
- [34] C. Flanagan, S. Freund, and M. Lifshin. Type inference for atomicity. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'05)*, pages 47–58, 2005.
- [35] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 338–349, 2003.
- [36] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 1–12, 2003.
- [37] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The *nesC* language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 1–11, 2003.
- [38] S. Ghemawat, K. Randall, and D. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 334–344, 2000.
- [39] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 13–25, 2003.
- [40] S. Guyer and C. Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Static Analysis Symposium (SAS'03)*, pages 214–236, 2003.
- [41] S. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005.

- [42] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on Model Checking Software (SPIN'00)*, pages 331–342, 2000.
- [43] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 168–181, 2003.
- [44] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 24–34, 2001.
- [45] T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 1–13, 2004.
- [46] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 54–61, 2001.
- [47] A. Kamil and K. Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, pages 185–199, 2005.
- [48] M. Lam, J. Whaley, B. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05)*, pages 1–12, 2005.
- [49] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

- [50] T. Lev-Ami, N. Immerman, T. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, pages 99–115, 2005.
- [51] O. Lhoták and L. Hendren. Jedd: a BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 158–169, 2004.
- [52] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, 2002.
- [53] L. Li and C. Verbrugge. A practical MHP information analysis for concurrent Java programs. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*, pages 194–208, 2004.
- [54] R. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
- [55] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391, 2005.
- [56] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*, pages 129–138, 1993.
- [57] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 346–358, 2006.
- [58] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, pages 476–490, 2005.

- [59] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 4th Annual Conference on Supercomputing (SC'91)*, pages 24–35, 1991.
- [60] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 1–11, 2002.
- [61] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, 2005.
- [62] G. Naumovich and G. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'98)*, pages 24–34, 1998.
- [63] G. Naumovich, G. Avrunin, and L. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99)*, pages 338–354, 1999.
- [64] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, pages 127–138, 2004.
- [65] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 167–178, 2003.
- [66] K. Olukotun and L. Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, 2005.

- [67] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '94)*, pages 324–340, 1994.
- [68] E. Pozniarsky and A. Schuster. Efficient on-the-fly data race detection in multi-threaded C++ programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 179–190, 2003.
- [69] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 320–331, 2006.
- [70] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 14–24, 2004.
- [71] M. Ronsse and K. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.
- [72] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 208–218, 2000.
- [73] A. Sasturkar, R. Agarwal, L. Wang, and S. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 83–94, 2005.
- [74] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP'97)*, pages 27–37, 1997.

- [75] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'89)*, pages 285–297, 1989.
- [76] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference (HVC'06)*, pages 166–182, 2006.
- [77] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 387–400, 2006.
- [78] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 59–76, 2005.
- [79] N. Sterling. WARLOCK - a static data race analysis tool. In *Proceedings of the Usenix Winter 1993 Technical Conference*, pages 97–106, 1993.
- [80] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3), 2005.
- [81] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 125–135, 1999.
- [82] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 334–345, 2006.
- [83] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 70–82, 2001.

- [84] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 115–128, 2003.
- [85] L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, pages 61–71, 2005.
- [86] L. Wang and S. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32(2):93–110, 2006.
- [87] J. Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, 2007.
- [88] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, 2004.
- [89] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 187–206, 1999.
- [90] R. Wilhelm, S. Sagiv, and T. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, pages 1–17, 2000.
- [91] A. Williams, W. Thies, and M. Ernst. Static deadlock detection for Java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 602–629, 2005.
- [92] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 221–234, 2005.

- [93] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, 2008.
- [94] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04)*, pages 145–157, 2004.