RedCard: Redundant Check Elimination For Dynamic Race Detectors Technical Report UCSC-SOE-13-05*

Cormac Flanagan[1] Stephen N. Freund[2]

[1] University of California at Santa Cruz
[2] Williams College

Abstract. Precise dynamic race detectors report an error if and only if an observed program trace exhibits a data race. They must typically check for races on all memory accesses to ensure that they catch all races and generate no spurious warnings. However, a race check for a particular memory access is guaranteed to be *redundant* if the accessing thread has already accessed that location within the same *release-free span*. A release-free span is any sequence of instructions containing no lock releases or other "release-like" synchronization operations, such as `notify` or `fork`.

We present a static analysis to identify redundant race checks by reasoning about memory accesses within release-free spans. In contrast to prior whole program analyses for identifying accesses that are always race-free, our redundant check analysis is span-local and can also be made method-local without any major loss in effectiveness. RedCard, our prototype implementation for the Java language, enables dynamic race detectors to reduce the number of run-time checks by close to 40% with no loss in precision.

We also present a complementary *shadow proxy* analysis for identifying when multiple memory locations can be treated as a single location by a dynamic race detector, again with no loss in precision. Combined, our analyses reduce the number of memory accesses requiring checks by roughly 50%.

1 Introduction

Multithreaded programs are prone to race conditions caused by unintended interference between threads, a problem exacerbated by the broad adoption of multi-core processors. A race condition occurs when two threads concurrently perform *conflicting* memory accesses that read or write the same location, where at least one access is a write. The order in which the conflicting accesses are performed may affect the program's subsequent state and behavior, likely with unintended or erroneous consequences. Such problems may arise only on rare interleavings, making them difficult to detect, reproduce, and eliminate.

* This work appears in ECOOP 2013.

The problems caused by data races have motivated much work on detecting races via static [1, 3, 5, 15, 23, 30, 40] or dynamic [10, 14, 31, 32, 35, 37, 43] analysis, as well as via post-mortem analyses [2, 9, 34]. In this paper, we focus on on-line dynamic race detectors, which detect races by monitoring a program as it executes. Dynamic race detectors typically use a broad notion of when two conflicting accesses are considered concurrent to maximize coverage, and conflicting accesses need not be performed at exactly the same time. Instead, a race condition is said to occur when there is no "synchronization dependence" between the two accesses, such as the dependence between a lock release by one thread and a subsequent lock acquire by a different thread. These various kinds of synchronization dependencies form a partial order over the instructions in the execution trace called the *happens-before relation* [26]. Two memory accesses are considered to be *concurrent* if they are not ordered by this relation.

Dynamic detectors may be classified by whether they are *precise* or *imprecise*. Precise analyses never produce false alarms. Instead, they compute an exact representation of the happens-before relation for the observed trace and report an error if and only if the observed trace has a race condition [18, 28, 32].[1] Despite the development of a variety of implementation techniques (including vector clocks [28, 32], epochs [18], accordion clocks [10], and others [14]), the overhead of precise dynamic race detectors can still be prohibitive.

A promising approach for improving the performance of precise dynamic race detectors is to use a static analysis to identify accesses that do not need to be checked at run time. Several prior analyses identify accesses that are guaranteed to be *race-free*, with good results [8, 17, 30, 38], but the more effective analyses are typically whole-program and quite expensive.

We focus instead on the orthogonal and more tractable property of identifying race checks that are guaranteed to be *redundant*, where a race check is redundant if ignoring that access during dynamic race checking leads to no missed races or false alarms. Interestingly, whereas verifying an access to be statically race-free typically requires information about multithreaded control flow, aliasing, and synchronization most readily computed via whole-program analysis, many race checks can be verified as redundant using more local information, specifically the locations the current thread has accessed since entering the current *release-free span*. Informally, a release-free span is a sequence of instructions containing no lock releases (or other synchronization operations such as `fork` that may similarly introduce an outgoing edge in the happens-before graph).

We present a static analysis for identifying and eliminating redundant race checks based on this notion. While others have explored removing limited forms of redundant checks in various imprecise detectors [8, 13] and in programs with structured parallelism [33], we focus exclusively on redundancy (independent of

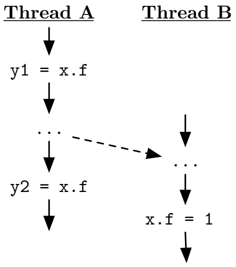
[1] Precise dynamic race detectors do not reason about all possible traces and may not identify races occurring on unobserved code paths. While full coverage is desirable, it comes at the cost of potential false alarms due to the undecidability of the halting problem. To avoid false alarms, precise race detectors thus focus on detecting race conditions only on the observed trace.

the dynamic race detection algorithm used) and tailor our analysis to be highly effective at reasoning about that notion. It primarily leverages local reasoning about memory accesses, aliasing, data invariants, and synchronization to eliminate a substantial amount of redundant checking for any dynamic race detector, with no loss in precision.

Redundant Check Elimination. A *release-free span* is a code fragment containing no operations that create out-going edges in the happens-before graph. Thus release-free spans may not contain lock releases (which create happens-before edges from the release to any subsequent acquires by other threads), forks (which create edges from the fork to the first step of the new thread), writes to volatiles (which create edges from the write to subsequent reads), and so on.

Spans exhibit a key property for our analysis: if an access to a memory location is in a race with a step by another thread, all previous accesses to that memory location within the current span will also be in a race with the other thread.

To illustrate why this property holds, suppose Thread A executes some span reading `x.f` twice while Thread B writes to `x.f`. If the write is in a race with the second read, then it must also be in a race with the first. If this were not the case, then the happens-before graph for the racy execution would have the form shown on the right, where the dashed line is present due to the synchronization operation ensuring that the write happens after the first read. However, this situation is impossible, because release-free spans contain no out-going edges to steps by other threads, and so the dashed line cannot exist. Thus it is sufficient for a dynamic race detector to check for races only on the first access to a memory location in each span. It may ignore any subsequent accesses without any loss in precision.



At a high level, our analysis records at each program point a context Π containing *available paths* describing memory locations previously accessed in the current release-free span, as well as other local state invariants we describe later. To illustrate this idea, consider the code to the right in which lines 1–3 form a release-free span. Our system verifies that `x.f` is an available path at line 3, and thus labels only the first access in the span as `Check` to indicate that the dynamic detector must examine that access. No paths are available after the span ends on line 4, and the detector must again check the first access to `x.f` in the new span.

```
1 synchronized(m) {
2   y = x.f^{Check};
3   y = x.f^{NoCheck};
4 }
5 y = x.f^{Check};
```

To support arrays, contexts may also include universally quantified paths, as in the code in Figure 1, which clears a two-dimensional array. The comments indicate the most salient context items inferred at various program points. Examining the inner loop on lines 3–6, the `Check` annotation for assignment “$\texttt{a[i]}^{\texttt{NoCheck}}\texttt{[j]}^{\texttt{Check}}\texttt{ = 0}$” on line 5 indicates that only a single check, on the access to the $\texttt{j}^{th}$ element of array `a[i]`, is required on each iteration of the

Figure 1: Redundant Check Elimination for Arrays.

```
1  // Π : ∅
2  for (int i = 0; i < a.length; i++) {
3    for (int j = 0; j < a[i]^Check.length; j++) {
4      // Π : a[i], ∀(k ∈ 0 to j). a[i][k]
5      a[i]^NoCheck[j]^Check = 0;
6    }
7    // Π : ∀(h ∈ 0 to i). ∀(j ∈ 0 to a[h].length). a[h][j]
8  }
9  // Π : ∀(i ∈ 0 to a.length). ∀(j ∈ 0 to a[i].length). a[i][j]
```

loop. Race checks on the access `a[i]` on 5 are redundant, because that location was previously checked, as indicated by its presence in the context on line 4. Removing these checks substantially reduces the number of run-time checks performed. The post-loop context on 9 shows that any subsequent accesses to this two-dimensional array within the current span would also be check-free.

Shadow Proxies. Our analysis also identifies memory locations for which no checks are ever required, which further reduces time and space overhead since no analysis (or *shadow*) state needs to be maintained for such locations. Our approach is based on the observation that accesses to different memory locations are often correlated. For example, a `Point` object may be used in such a way that whenever its `x` field is accessed, its `y` field is accessed *in the same span*. We say that `y` is a shadow proxy for `x` in this case, and any race on `x` naturally implies that there is a race on `y`. A dynamic race detector will thus still detect the presence of a race, even if all accesses to `x` are ignored. Our analysis also identifies shadow proxy relationships between array elements.

RedCard. We have implemented our analysis in RedCard (Redundant Checks for Race Detectors) for Java bytecode programs. On a collection of benchmarks, RedCard reduced the number of run-time race condition checks required by a precise detector by roughly 40%. When configured to also infer shadow proxies, RedCard reduced the number of checks by close to 50%. Eliminating these redundant checks in the FastTrack dynamic race detector [18] improved its running-time by about 25%.

A number of other tools, such as Chord [30], leverage global may-happen-in-parallel or other flow-insensitive analyses to reason about conflicting accesses. This can be quite effective at finding unnecessary checks in some programs, but typically requires more expensive and less scalable global reasoning. We compare RedCard to Chord-like analyses in more detail in our experimental validation.

Contributions. In summary, this paper:

- defines a notion of a redundant check for precise dynamic race detectors, describes an analysis to identify redundant checks as an effect system for an idealized language, and proves this analysis is correct (Sections 2 and 3);

Figure 2: RedJava **Syntax**

$$
\begin{array}{rll}
\mathcal{P} \in \mathit{Program} & ::= & \overline{D} \;\; s_1 \| \ldots \| s_n \\
D \in \mathit{Defn} & ::= & \texttt{class}\ C\ \texttt{\{}\ \overline{f}\ \ \overline{\mathit{meth}}\ \texttt{\}} \\
\mathit{meth} \in \mathit{Method} & ::= & m\texttt{(}\overline{x}\texttt{)}\ \mathit{spec}\ \texttt{\{}\ s\texttt{;}\ \texttt{return}\ r\ \texttt{\}} \\
 & & \\
s \in \mathit{Stmt} & ::= & \texttt{skip} \mid s; s \mid \texttt{if}\ \mathit{be}\ s\ s \mid \texttt{while}\ \mathit{be}\ s \mid x\ \texttt{=}\ e \mid x\ \texttt{=}\ \texttt{new}\ C \\
 & & \mid y.f^k\ \texttt{=}\ x \mid x\ \texttt{=}\ y.f^k \mid x\ \texttt{=}\ y.m\texttt{(}\overline{z}\texttt{)} \mid \texttt{acq}\ x \mid \texttt{rel}\ x \\
 & & \\
e \in \mathit{Expr} & ::= & x \mid v \mid \ldots \\
\mathit{be} \in \mathit{BoolExpr} & \subseteq & \mathit{Expr} \\
v \in \mathit{Value} & ::= & \rho \mid \texttt{true} \mid \texttt{false} \mid \texttt{null} \mid \ldots \\
k \in \mathit{CheckOption} & ::= & \texttt{Check} \mid \texttt{NoCheck} \\
 & & \\
C \in \mathit{ClassName} & f \in \mathit{FieldName} & m \in \mathit{MethodName} \\
x, y, r \in \mathit{Var} & \rho \in \mathit{Address} &
\end{array}
$$

- extends the core analysis to handle arrays (Section 4) and to identify shadow proxies, which characterize memory locations that can be ignored by race detectors entirely, with no loss in precision (Section 5);
- describes our RedCard system for inferring redundant accesses in Java programs (Section 6); and
- shows that, on a collection of benchmarks, RedCard reduces the number of access checks required by a precise detector by close to 50%, leading to a roughly 25% speedup in the FastTrack race detector (Section 7).

2 RedJava Language and Semantics

We formalize our analysis for the idealized language RedJava, a multithreaded subset of Java summarized in Figure 2. A RedJava program $\mathcal{P}$ is a sequence of class definitions $\overline{D}$ together with a sequence of statements $s_1 \| \ldots \| s_n$. At run time, the statements in $s_1 \| \ldots \| s_n$ are evaluated concurrently by multiple threads.

Each definition associates a class name C with a collection of field and method declarations. Field declarations are simply names and contain no type information. (We assume that standard typing requirements are verified for RedJava programs via a separate analysis. Enforcing a traditional typing discipline is orthogonal to the concerns of this paper and omitted for simplicity.) A method declaration "$m\texttt{(}\overline{x}\texttt{)}\ \mathit{spec}\ \texttt{\{}\ s\texttt{;}\ \texttt{return}\ r\ \texttt{\}}$" defines a method m with parameters $\overline{x}$ and statement body s. The method returns the value stored in variable r. The variable `this` is implicitly bound to the receiver in the method body. We assume that all methods have unique names to avoid type-based method resolution. Methods also contain specifications, as described below.

RedJava statements are expressed in a low-level language somewhat analogous to JVM bytecode. Statement forms include sequential composition, conditionals, while loops, and method calls. Local variables, which are not explicitly declared, can be mutated via the assignment statement "`x = `e". We leave the set of expressions e intentionally unspecified but assume that they range over at least `null`, boolean values, variable identifiers, and object addresses.

The object allocation statement "`x = new C`" assigns a freshly allocated C object to variable `x`, where all fields of that object are initialized to `null`. Field read ($\texttt{x = y.f}^k$) and write ($\texttt{y.f}^k\texttt{ = x}$) statements includes a check tag k, which is `Check` if the dynamic race detector should verify it for race-freedom and `NoCheck` if the dynamic race detector should skip verifying that access. As in Java, each object x has a corresponding mutual exclusion lock with operations `acq` x and `rel` x.

A program $\overline{D}\ s_1 \| \ldots \| s_n$ executes by evaluating the statements $s_1, \ldots, s_n$ in concurrent threads. The Appendix formalizes a small-step semantics describing evaluation as a relation $\overline{D} \vdash \Sigma \rightarrow^a \Sigma'$, where the run-time state Σ stores a heap of dynamically allocated objects; and Σ' is the same heap updated with the effects of this step. The *Action* a records any shared-memory or synchronization operation performed by the step. For example, the action $t\!:\!\texttt{acc}(\rho.f^{\texttt{Check}})$ denotes that the thread identifier $t \in \mathit{Tid}$ performed a checked access to the field `f` of the object at address ρ. The special action $t\!:\!\epsilon$ denotes a step that has no heap effect.

$$\begin{array}{rcl} u, t \in \mathit{Tid} & ::= & 1 \ \mid\ 2 \ \mid\ \ldots \\ a, b \in \mathit{Action} & ::= & t\!:\!\texttt{acc}(\rho.f^k) \ \mid\ t\!:\!\texttt{acq}(\rho) \ \mid\ t\!:\!\texttt{rel}(\rho) \ \mid\ t\!:\!\epsilon \\ \alpha \in \mathit{Trace} & ::= & \overline{\mathit{Action}} \end{array}$$

The relation $\overline{D} \vdash \Sigma \rightarrow^\alpha \Sigma'$ is the reflexive transitive closure of the single-step relation and formalizes the behavior of a *Trace* $\alpha = a_1 \cdot a_2 \cdots a_n$. The initial state Σ_0 contains a freshly-allocated object for each free variable in $s_1 \| \ldots \| s_n$. Those objects may be referenced by multiple threads.

2.1 Race Conditions and Dynamic Race Detection

The *happens-before* relation for trace α is the smallest reflexive, transitive relation $<_\alpha$ on α such that $a <_\alpha b$ if a occurs before b in α and either: a and b are performed by the same thread; or a releases some lock and b acquires that lock.

Two operations are *concurrent* if they are not ordered by the happens-before relation, and two accesses *conflict* if they read or write to the same location $\rho.f$. Our definition of conflicting accesses implies that two reads may conflict, which simplifies our formal development. We address commuting read operations in Section 6. A trace has a *race condition* if it has a pair of concurrent conflicting accesses. Moreover, a trace has a *detected race condition* if it has a pair of concurrent conflicting accesses that are both marked as `Check`.

We now consider which accesses in a trace require checks. A *release-free span*, or simply a *span*, is the sequence of instructions by a thread between two release statements. A trace is *well-formed* if each unchecked access t : $\texttt{acc}(\rho.f^{\texttt{NoCheck}})$ is preceded by a checked access $t\!:\!\texttt{acc}(\rho.f^{\texttt{Check}})$ in the same span, under the assumption the trace prefix up to the unchecked access is race-free.

Figure 3: Contexts, Path Sets, and Modifies Sets.

$$
\begin{array}{lll}
\mathit{\Pi} \in \mathit{Context} & ::= & \overline{\pi} \\
\pi \in \mathit{ContextItem} & ::= & p \mid c \mid \mathit{be} \\
c \in \mathit{AliasConstraint} & ::= & x = p
\end{array}
\qquad
\begin{array}{lll}
P, Q \in \mathit{PathSet} & ::= & \overline{p} \\
p \in \mathit{Path} & ::= & x.f \\
K \in \mathit{ModifiesSet} & ::= & \overline{\kappa} \\
\kappa & ::= & f \mid \textsc{All}
\end{array}
$$

$$
\begin{array}{l}
\texttt{y1 = x.f}^{\texttt{Check}}\texttt{;} \\
\texttt{z1 = y1.g}^{\texttt{Check}}\texttt{;} \\
\texttt{y2 = x.f}^{\texttt{NoCheck}}\texttt{;} \\
\texttt{z2 = y2.g}^{\texttt{NoCheck}}\texttt{;}
\end{array}
$$

To motivate this race-free assumption, consider the code fragment to the right. We would like to annotate the last read `y2.g` as `NoCheck`, arguing that `y1` and `y2` are aliases, and that `y1.g` was previously read. However, concurrent racy writes to `x.f` could cause `y1` and `y2` to differ. The race-free prefix assumption precludes this possibility and enables the unchecked read of $\texttt{y2.g}^{\texttt{NoCheck}}$.

The following theorem, which is proved in the companion technical report, shows that the checked accesses in well-formed traces are sufficient to guarantee that, for any trace with one or more race conditions, at least one of those races will be detected. In practice, our implementation detects all races on all of our benchmarks, since the above corner case in which one race masks another race is extremely rare.

Theorem 1 (Race Detection). *Any well-formed trace has a race condition if and only if it has a detected race condition.*

3 Redundant Check Elimination

We now develop a static analysis for identifying which accesses are redundant (that is, race-free race under the assumption that previous accesses are race-free). For presentation purposes, we describe our analysis as a decision procedure for verifying `NoCheck` annotations, but this decision procedure naturally maps to an inter-procedural least-fixed point algorithm for inferring `NoCheck` annotations in our implementation, as described in Section 6.

Our analysis tracks a context $\mathit{\Pi}$ that includes paths p of the form $x.f$ that have already been accessed in the current span.[2] In the simple case, redundant check elimination can be accomplished by standard compiler optimizations for redundant load elimination (see, *e.g.* [4, 24, 39]), which may remove the second redundant read of $x.f$ entirely. However, redundant check elimination applies in more general cases, for example when iterating through an array for a second time in a span, since the array contents likely would not fit in the register file, or on a read that occurs after different writes on different control-flow paths.

To help identify redundant checks, the context includes must-alias information, such as the equality `y1` = `y2` from the earlier example in Section 2.1. Finally the context includes boolean constraints over local variables, which additionally aid in reasoning about both aliasing and array accesses, as discussed below.

[2] We discuss longer access paths, such as `a.f[i]`, in Section 6.

Figure 4: Analysis (We assume w is fresh in all rules.)

$$\boxed{\overline{D} \vdash s : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

$$\begin{array}{lrcl}
[\textsc{Skip}] & \overline{D} \vdash \texttt{skip} : \mathit{\Pi} & \rightsquigarrow & \mathit{\Pi} \\
[\textsc{Acq}] & \overline{D} \vdash \texttt{acq}\ x : \mathit{\Pi} & \rightsquigarrow & \mathit{\Pi} \\
[\textsc{Rel}] & \overline{D} \vdash \texttt{rel}\ x : \mathit{\Pi} & \rightsquigarrow & \mathit{\Pi} \setminus \textsc{All} \\
[\textsc{Assign}] & \overline{D} \vdash x\ \texttt{=}\ e : \mathit{\Pi} & \rightsquigarrow & \mathit{\Pi}[x := w] \cup \{x = e[x := w]\} \\
[\textsc{New}] & \overline{D} \vdash x\ \texttt{=}\ \texttt{new}\ C : \mathit{\Pi} & \rightsquigarrow & \mathit{\Pi}[x := w]
\end{array}$$

[Read]

$$\frac{\begin{array}{c} k = \texttt{NoCheck} \ \Rightarrow\ \mathit{\Pi} \vdash y.f \\ p = (y.f)[x := w] \qquad \mathit{\Pi}' = \mathit{\Pi}[x := w] \cup \{x = p, p\} \end{array}}{\overline{D} \vdash x\ \texttt{=}\ y.f^k : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

[Write]

$$\frac{\begin{array}{c} k = \texttt{NoCheck} \ \Rightarrow\ \mathit{\Pi} \vdash y.f \\ \mathit{\Pi}' = (\mathit{\Pi} \setminus f) \cup \{x = y.f, y.f\} \end{array}}{\overline{D} \vdash y.f^k\ \texttt{=}\ x : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

[If]

$$\frac{\begin{array}{c} \overline{D} \vdash s_1 : \mathit{\Pi} \cup \{be\} \rightsquigarrow \mathit{\Pi}_1 \\ \overline{D} \vdash s_2 : \mathit{\Pi} \cup \{\neg be\} \rightsquigarrow \mathit{\Pi}_2 \end{array}}{\overline{D} \vdash \texttt{if}\ be\ s_1\ s_2 : \mathit{\Pi} \rightsquigarrow (\mathit{\Pi}_1 \sqcap \mathit{\Pi}_2)}$$

[While]

$$\frac{\begin{array}{c} \mathit{\Pi} \sqsubseteq \mathit{\Pi}_{inv} \qquad \mathit{\Pi}' \sqsubseteq \mathit{\Pi}_{inv} \\ \overline{D} \vdash s : (\mathit{\Pi}_{inv} \cup \{be\}) \rightsquigarrow \mathit{\Pi}' \end{array}}{\overline{D} \vdash \texttt{while}\ be\ s : \mathit{\Pi} \rightsquigarrow (\mathit{\Pi}_{inv} \cup \{\neg be\})}$$

[Seq]

$$\frac{\begin{array}{c} \overline{D} \vdash s_1 : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'' \\ \overline{D} \vdash s_2 : \mathit{\Pi}'' \rightsquigarrow \mathit{\Pi}' \end{array}}{\overline{D} \vdash s_1; s_2 : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

[Call]

$$\frac{\begin{array}{c} m(\overline{z'})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \texttt{\{}\ \ldots\ \texttt{return}\ r\ \texttt{\}} \in \overline{D} \\ \theta = [\overline{z' := z}, \texttt{this} := y, r := x] \qquad \forall p \in \theta(P).\ \mathit{\Pi} \vdash p \end{array}}{\overline{D} \vdash x\ \texttt{=}\ y.m(\overline{z}) : \mathit{\Pi} \rightsquigarrow (\mathit{\Pi} \setminus K)[x := w] \cup \theta(Q)}$$

[Method]

$$\frac{\begin{array}{c} m \text{ is unique in } \overline{D} \qquad s \text{ does not mutate } \texttt{this}, \overline{x} \\ K \text{ contains all fields that may be modified while evaluating } s \\ K \text{ contains } \textsc{All} \text{ if a lock may be released while evaluating } s \\ \mathrm{FV}(P) \subseteq \{\texttt{this}, \overline{x}\} \qquad \mathrm{FV}(Q) \subseteq \{\texttt{this}, \overline{x}, r\} \\ \overline{D} \vdash s : P \rightsquigarrow Q' \qquad Q' \sqsubseteq Q \end{array}}{\overline{D} \vdash m(\overline{x})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \texttt{\{}\ s; \texttt{return}\ r\ \texttt{\}}}$$

$$\boxed{\overline{D} \vdash meth \qquad \vdash \overline{D} \qquad \vdash \overline{D}\ s_1 \| \ldots \| s_n}$$

[Class]

$$\frac{\forall meth \in \overline{meth}.\ \overline{D} \vdash meth}{\overline{D} \vdash \texttt{class}\ C\ \texttt{\{}\ \overline{f}\ \ \overline{meth}\ \texttt{\}}}$$

[Declarations]

$$\frac{\forall D \in \overline{D}.\ \overline{D} \vdash D}{\vdash \overline{D}}$$

[Program]

$$\frac{\vdash \overline{D} \qquad \forall i \in 1..n.\ \overline{D} \vdash s_i : \emptyset \rightsquigarrow \mathit{\Pi}}{\vdash \overline{D}\ s_1 \| \ldots \| s_n}$$

$$\boxed{\mathit{\Pi} \setminus K,\ \mathit{\Pi} \setminus \kappa}$$

$$\begin{array}{rcl}
\mathit{\Pi} \setminus \{\kappa_1, \ldots, \kappa_n\} & = & \mathit{\Pi} \setminus \kappa_1 \setminus \ldots \setminus \kappa_n \\
\mathit{\Pi} \setminus f & = & \{\ p \in \mathit{\Pi}\ \} \cup \{\ be \in \mathit{\Pi}\ \} \cup \{\ (y = p) \in \mathit{\Pi} \ \mid\ p \neq x.f\ \ \} \\
\mathit{\Pi} \setminus \textsc{All} & = & \{\ be \in \mathit{\Pi}\ \}
\end{array}$$

To facilitate modular reasoning, each method carries a specification

$$spec \ \in\ MethodSpec \ = \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K$$

where the *PathSet* P denotes the paths that must be in the context at any call site to the method, which enables inter-procedural check elimination. The PathSet Q denotes paths that are available at the end of the method body, and the *ModifiesSet* K denotes fields that may be modified by the method body. (See Figure 3.) The ModifiesSet K also contains the special token All if the method contains a lock release or other span-ending operation.

3.1 Type System

The core of our analysis is the set of rules in Figure 4 defining the judgment $\overline{D} \vdash s : \Pi \rightsquigarrow \Pi'$, where the context Π denotes available paths and constraints that hold in the pre-state of statement s, and context Π' similarly characterizes the post-state of s. The definitions $\overline{D}$ are included to provide access to class declarations. The complete set of statement typing rules, as well as rules for additional judgments to check declarations and programs, appear in Figure 4. We describe the most salient rules below.

[Assign] Assigning to a local variable x may affect constraints or paths containing x. Simply removing all such elements mentioning x from Π would be overly conservative. Instead, we introduce a class of *Skolem variables* that are implicitly existentially quantified. The post-context for a statement x `=` e from a pre-context Π is then computed to be $\Pi[x := w] \cup \{x = e[x := w]\}$, where the fresh implicitly existentially quantified variable w captures the pre-assignment value of x. (The substitution $\Pi[x := w]$ replaces all occurrences of x with w in Π, and similarly for $e[x := w]$.)

To illustrate this rule, consider the code fragment to the right. The context prior to the assignment to `z` contains `a` $=$ `z.f` and `b` $=$ `z.f`, which implies `a` and `b` are aliases. After the assignment to `z`, the context will contain `a` $= w.$`f` and `b` $= w.$`f` for some fresh w. Thus we may still prove `a` and `b` are aliases and the last access is redundant, even though `a` and `b` are no longer equal to `z.f`.

```
a = z.f$^{\texttt{Check}}$;
b = z.f$^{\texttt{NoCheck}}$;
z = y;
t1 = a.g$^{\texttt{Check}}$;
t2 = b.g$^{\texttt{NoCheck}}$;
```

Note that this rule adds the equality $x = e[x := w]$ to Π, where e may be any expression. The most useful equality constraints for reasoning about field accesses have the form $x = y$, but we will see in Section 4 that constraints over more complex scalar expressions are crucial for reasoning about array accesses.

[Read] For the read statement x `=` $y.f^k$, we verify that $y.f$ has already been accessed in the current span if $k =$ `NoCheck`. To do this, we could simply require that $y.f \in \Pi$. However, this syntactic notion of membership does not take into account aliasing or other information. Thus we introduce the *context implication* relation $\Pi \vdash y.f$ to indicate that the elements in context Π imply that $y.f$ has been accessed, perhaps via an aliased name:

$$\Pi \vdash y.f \quad \text{iff} \quad \Pi \vdash z = y \wedge z.f \in \Pi$$

Any sound decision procedure may be used to implement the relation $\Pi \vdash z = y$. In our implementation, we leverage the SMT solver Z3 [11], making sure that the translation into Z3's input language is appropriately conservative. Below, we use a generalized context implication relation $\Pi \vdash \pi$ to express that the context Π implies the context item π, which may be a path, alias constraint, or a boolean constraint.

The substitution $y.f[x := w]$ in this rule ensures that the proper elements are added to post-context Π' in the case where $y = x$.

[Write] For the write statement $y.f^k$ `=` x, we verify that $y.f$ has already been accessed in the current span if $k = \texttt{NoCheck}$. To compute the post-context Π', we remove all equalities referring to the f field of an object via the operation $\Pi \setminus f$ defined in Figure 4. After the assignment, we include both the equality $x = y.f$ and the available path $y.f$ in Π'.

[If] The rule for conditionals merges the post-contexts of the *then* and *else* branches via the meet operator

$$\Pi_1 \sqcap \Pi_2 \quad = \quad \{\ \pi \in \Pi_1 \cup \Pi_2 \ \mid \ \Pi_1 \vdash \pi \text{ and } \Pi_2 \vdash \pi\ \}$$

This operator computes the largest set of facts in $\Pi_1 \cup \Pi_2$ that are provable from both Π_1 and Π_2. Those contexts may refer to different Skolem variables, but this rule does not attempt to unify them in any way. Thus, any π in $\Pi_1 \sqcap \Pi_2$ will only refer to Skolem variables appearing in both Π_1 and Π_2.

[While] The rule for loops identifies an appropriate loop invariant context Π_{inv} that is implied by the pre-state of the loop and also preserved by each iteration of the loop body. Implication between contexts is defined as

$$\Pi_1 \sqsubseteq \Pi_2 \quad \text{iff} \quad \forall \pi \in \Pi_2.\ \Pi_1 \vdash \pi$$

Our implementation uses a form of Cartesian predicate abstraction [20,22] to heuristically compute appropriate loop invariants, as described in Section 6. It synthesizes initial candidate invariants based on loop induction variable and bounds information extracted from the code and pattern matching for common idioms.

[Call] This rule for a method call x `=` $y.m(z_{1..n})$ first ensures that there is a corresponding method definition

$$m(\overline{z'})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \{\ s; \texttt{return}\ r\ \}$$

and creates a substitution θ to map `this`, the formal parameters $\overline{z'}$, and the result variable r to the corresponding variables y, $\overline{z}$, and x at the call site. The PathSet precondition P describes paths that must be available at call sites, and this rule checks that each path $p \in \theta(P)$ is entailed by Π.

The ModifiesSet K contains fields assigned to while evaluating the body of m (either directly, or indirectly while evaluating a nested method call), as well as a special token All if m performs a release or other span-ending operation. (That requirement is enforced in [Method].) The operation $\Pi \setminus K$ removes any invalidated constraints from the context Π. We also replace occurrences of x in Π as in the previous rules and add the paths from the method's post-context Q, yielding the final context $\Pi' = (\Pi \setminus K)[x := w] \ \cup\ \theta(Q)$.

[Acq] Acquires do not change the current context.

[Rel] Once a lock is released, we cannot assume any location is race-free or any aliasing constraint still holds. Thus, we remove all aliasing information and available paths from the context, leaving just the boolean expression constraints, which only refer to local variables.

3.2 Correctness

We now sketch the correctness guarantees provided by our analysis. The Appendix contains the full details of our formal development. We first note that the semantics exhibits the standard preservation property that evaluation preserves well-typing under an appropriate extension of the type system to run-time states, denoted $\overline{D}, \alpha \vdash \Sigma$. This judgment ensures that the run-time state is consistent with the statically-computed contexts for each thread's code. The judgment includes the trace α to provide information about previous accesses performed within currently active spans.

Our first main correctness result is a preservation theorem showing that if our analysis can verify the `NoCheck` annotations in a run-time state Σ with execution trace α, then it can also verify any successor state Σ' with respect to an appropriately extended trace, provided that the extended trace is race-free.

Theorem 2 (Preservation). *If $\overline{D}; \alpha \vdash \Sigma$ and $\Sigma \longrightarrow^{a} \Sigma'$ and $\alpha \cdot a$ is race-free, then $D; (\alpha \cdot a) \vdash \Sigma'$.*

Based on this theorem, we then show that any verified program state Σ generates only well formed traces α.

Theorem 3 (Soundness). *If $\overline{D}; \epsilon \vdash \Sigma$ and $\Sigma \longrightarrow^{\alpha} \Sigma'$, then α is well-formed.*

Thus, by Theorem 3, a race detector can safely ignore `NoCheck` accesses in any verified program, with no loss in precision.

4 Arrays

In order to better optimize array-intensive programs, we extend our analysis to support universally quantified paths, as illustrated by the following code fragment to clear an array, as well as computed analysis contexts:

```
i = 0;
// $\Pi$ : $\texttt{i} = 0$
while (i < a.length) {
  // $\Pi$ : $\forall(j \in 0$ to i). a[$j$], i $<$ a.length
  a[i]$^{\texttt{Check}}$ = 0;
  // $\Pi$ : $\forall(j \in 0$ to i). a[$j$], a[i], i $<$ a.length
  i = i + 1;
  // $\Pi$ : $\forall(j \in 0$ to i$'$). a[$j$], a[i], i $=$ i$'$ $+ 1$, i$'$ $<$ a.length
}
// $\Pi$ : $\forall(j \in 0$ to a.length). a[$j$], i $\geq$ a.length
```

We wish to infer that after the loop terminates, all array elements have been accessed, which we capture as the *universally quantified path*

$$\forall j \in (0 \texttt{ to a.length}).\ \texttt{a[}j\texttt{]}$$

stating that array elements $\texttt{a[0]}, \texttt{a[1]}, \ldots, \texttt{a[a.length}-\texttt{1]}$ have been accessed. Verifying this loop post-condition naturally requires a corresponding loop invariant path

$$\forall j \in (0 \; \texttt{to i}).\ \texttt{a[}j\texttt{]}$$

and also a richer constraint language to capture relevant invariants on indexed variables. For example, on entry to the loop, the equality $\texttt{i} = 0$ ensures that the loop invariant holds initially:

$$\{\ \texttt{i} = 0\ \} \Rightarrow \{\ \forall j \in (0 \; \texttt{to i}).\ \texttt{a[}j\texttt{]}\ \}$$

Moreover, the invariant holds on the loop's back edge due to the following implication, where the Skolem variable i' refers to the value of `i` before the increment:

$$\{\ \forall j \in (0 \; \texttt{to}\; i').\ \texttt{a[}j\texttt{]}, \texttt{a[}i'\texttt{]},\ \texttt{i} = i' + 1\ \} \;\Rightarrow\; \{\ \forall j \in (0 \; \texttt{to i}).\ \texttt{a[}j\texttt{]}\ \}$$

Finally, on loop termination, we verify:

$$\{\ \forall j \in (0 \; \texttt{to i}).\ \texttt{a[}j\texttt{]}, \texttt{i} \geq \texttt{a.length}\ \} \;\Rightarrow\; \{\ \forall j \in (0 \; \texttt{to a.length}).\ \texttt{a[}j\texttt{]}\ \}$$

Figure 5 formalizes the extended language of quantified paths and type rules used by our analysis. Paths now include array accesses $x\texttt{[}z\texttt{]}$ as well as field accesses. Paths may also include an enclosing quantification $(\forall i \in r.\ \bullet)$ over a strided range r of the form "e_{start} $\texttt{to}$ e_{end} $\texttt{by}$ e_{step}", which represents the set of indices $\{\ i \in [e_{start}, e_{end}) \mid (i - e_{start}) \mod e_{step} = 0\ \}$. (We abbreviate the strided range "e_{start} $\texttt{to}$ e_{end} $\texttt{by}$ 1" as "e_{start} $\texttt{to}$ e_{end}".)

The rule [Array Write] for an array assignment $y\texttt{[}z\texttt{]}\ \texttt{=}\ x$ uses the operation $\mathit{\Pi} \setminus$ Array, defined below, to remove from $\mathit{\Pi}$ any information dependent on array values. We also include Array as a possible ModifiesSet component κ in method specifications.

$$\begin{array}{rcl} \kappa & ::= & f \;\mid\; \textsc{All} \;\mid\; \textsc{Array} \\ \mathit{\Pi} \setminus \textsc{Array} & = & \{\ p \in \mathit{\Pi}\ \} \;\cup\; \{\ be \in \mathit{\Pi}\ \} \;\cup\; \{\ (x = p) \in \mathit{\Pi} \;\mid\; p \neq y\texttt{[}e\texttt{]}\ \} \end{array}$$

This treatment of array writes is quite coarse since an assignment to any array eliminates facts about all arrays in the program. In practice, we use a type-based rule that, when an array of type C is modified, eliminates only facts dependent on values stored in arrays of type C. This refinement works well in practice, but further refinements based on more precise points-to information could be used in cases where it proves insufficient.

5 Shadow Proxies

Identifying redundant checks can improve the performance of dynamic race detectors, but it does not directly reduce the memory overhead of keeping analysis (or *shadow*) state for each memory location. Various race detectors [32, 43] have explored using one shadow state per object or array, but this approach may produce spurious warnings.

Figure 5: RedJava **Extensions to Support Arrays.**

$$
\begin{array}{rcll}
s & \in & \mathit{Stmt} & ::= \ \ldots \mid x\ \texttt{=}\ \texttt{new}\ C\texttt{[}z\texttt{]} \mid x\ \texttt{=}\ y\texttt{[}z\texttt{]}^{k} \mid y\texttt{[}z\texttt{]}^{k}\ \texttt{=}\ x \\[1ex]
p & \in & \mathit{Path} & ::= \ x.f \mid x\texttt{[}e\texttt{]} \\
qp & \in & \mathit{QuantifiedPath} & ::= \ p \mid \forall i \in r.\ qp \\
r & \in & \mathit{StridedRange} & ::= \ e\ \texttt{to}\ e\ \texttt{by}\ e \\[1ex]
\pi & \in & \mathit{ContextItem} & ::= \ qp \mid c \mid be
\end{array}
$$

$$\boxed{\overline{D} \vdash s : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

[Array Read]

$$\frac{k = \texttt{NoCheck} \ \Rightarrow\ \mathit{\Pi} \vdash y\texttt{[}z\texttt{]} \qquad p = (y\texttt{[}z\texttt{]})[x := w] \qquad \mathit{\Pi}' = \mathit{\Pi}[x := w] \cup \{x = p, p\}}{\overline{D} \vdash x\ \texttt{=}\ y\texttt{[}z\texttt{]}^{k} : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

[Array Write]

$$\frac{k = \texttt{NoCheck} \ \Rightarrow\ \mathit{\Pi} \vdash y\texttt{[}z\texttt{]} \qquad \mathit{\Pi}' = (\mathit{\Pi} \setminus \textsc{Array}) \cup \{x = y\texttt{[}z\texttt{]}, y\texttt{[}z\texttt{]}\}}{\overline{D} \vdash y\texttt{[}z\texttt{]}^{k}\ \texttt{=}\ x : \mathit{\Pi} \rightsquigarrow \mathit{\Pi}'}$$

We now explore how to reduce the number of shadow states via static analysis, while still preserving precision. Our approach is based on the observation that accesses to different memory locations are often correlated. For example, an object ρ of type C with fields f and g may be used in such a way that whenever $\rho.g$ is accessed, $\rho.f$ will also be accessed *in the same span.* Thus any data race on $\rho.g$ naturally implies there is a race on $\rho.f$. In this case, we say that $\rho.f$ is a *proxy location* for $\rho.g$, and we say that f is a *proxy field* for g, written $f \succ g$, if that proxy relationship holds for all C objects.

If $f \succ g$, then a race checker may forego allocating shadow state for g fields and ignore all accesses to them while still providing the following guarantees for all objects ρ:

- If no races are found on $\rho.f$, then both $\rho.f$ and $\rho.g$ are race-free.
- No races will be reported on $\rho.g$, since those accesses are not checked.
- A detected race on $\rho.f$ implies that there *is* a race on $\rho.f$ and that there *may be* a race on $\rho.g$.

Note that if both $f \succ g$ and $g \succ f$ (*i.e.*, f and g are always accessed together), then the third guarantee may be strengthened to:

- A detected race on $\rho.f$ implies that there are races on *both* $\rho.f$ and $\rho.g$.

However, the weaker asymmetric requirement $f \succ g$ enables more check elimination and memory footprint reduction while still providing the useful correctness guarantee of always detecting a data race on any trace containing one.

Although space limitations prevent a full discussion, we summarize type system extensions for identifying proxy fields in the rest of this section. The key typing requirement is that $f \succ g$ if and only if, for each access instruction $x\ \texttt{=}\ y.g^{k}$ or $y.g^{k}\ \texttt{=}\ x$, either

- $\mathit{\Pi} \vdash y.f$, where $\mathit{\Pi}$ is the instruction's computed context; or
- $\mathit{\Pi}' \vdash y.f$, where $\mathit{\Pi}'$ is the computed context for some instruction in the same span that post-dominates the access to g.

The first clause captures cases when f is accessed before g, and the second captures the converse. If these requirements hold, all race checks for g are redundant. In our Java implementation, we compute the post-dominator relation assuming all unchecked exceptions, such as `NullPointerException`s, are errors in the source code and guarantee no loss in precision only on error-free traces.

Array entries may also have proxies. Location $\rho\texttt{[}i\texttt{]}$ is a proxy location for $\rho\texttt{[}j\texttt{]}$ if $\rho\texttt{[}i\texttt{]}$ is accessed in every span accessing $\rho\texttt{[}j\texttt{]}$. We say that i is a *proxy index* for j if that requirement holds for all array accesses in a program. Distinguishing arrays by allocation site (via, for example, a standard points-to analysis) enables a more precise definition: $i \succ_l j$ if i is a proxy index for j for all arrays allocated at source location l. The key typing requirement for arrays is that $i \succ_l j$ if and only if, for each access instruction $x \texttt{ = } y\texttt{[}z\texttt{]}^k$ or $y\texttt{[}z\texttt{]}^k \texttt{ = } x$ such that y may be an array allocated at source location l and z's value may be j, either

- $\Pi \vdash y\texttt{[}i\texttt{]}$, where Π is the instruction's computed context; or
- $\Pi' \vdash y\texttt{[}i\texttt{]}$, where Π' is the computed context for some instruction in the same span that post-dominates the array access.

Given this definition, the race check on an access $y\texttt{[}z\texttt{]}$ is redundant if, for every possible value j for z and allocation site l for y, there exists an i such that $i \succ_l j$.

Typically, shadow proxies for all indices of an array can be summarized quite succinctly. For example, the relation $\forall i.\ (i \text{ div } 4) * 4 \succ_l i$ indicates that all arrays allocated at l are divided into chunks of four elements, where the first index in each chunk is a proxy for the other three indexes in the chunk. Any array access instruction guaranteed not to touch one of the proxies may be tagged with `NoCheck`. Moreover, a race detector can use the shadow location maintained for each chunk's proxy when checking accesses to any elements in the chunk, reducing the number of required shadow locations for an array of n elements from n to $n/4$. Similar reductions in checking and memory requirements are possible for many different proxy relations, several of which are described below.

The shadow proxy analysis requires information about each span. However, one may be able to identify many proxies without examining all spans: for example, proxies among private fields can be found by examining only a single class at a time. We could also augment our system with simple "proxy specifications" on classes, which could then be subsequently verified by a modular analysis. We hope to explore these items in more detail in the future.

6 Implementation

We have implemented our analysis for the full Java language as part of a tool called RedCard. This analysis reads in bytecode programs and labels each field and array access as either `NoCheck` or `Check`. It outputs a list of the accesses marked `NoCheck`, and that list may then be used to optimize any dynamic race detector. Transforming our type system to the full Java language and an inference algorithm for bytecode is mostly straightforward. We describe the most interesting implementation details below.

We implemented RedCard in the WALA analysis infrastructure [41], which represents method bodies as control flow graphs over instructions in SSA form. The tool analyzes all methods appearing in a call graph built by WALA using 0-CFA. RedCard implements the relation $\Pi \vdash \pi$ via the Z3 SMT Solver [11].

For each instruction in each method, we compute a context Π via dataflow analysis. All contexts begin as the special context Top, where $\Pi \sqcap \textsc{Top} = \Pi$ for all contexts Π. The analysis then uses a meet operator and transfer functions based on the system presented in Section 3.1 to compute the maximal fixed point solution for the contexts for each instruction. RedCard handles all basic synchronization operations present in Java, including locks, volatile variables, fork/join, wait/notify, and so on. RedCard then identifies each memory access instruction for which its context implies a check is redundant.

Intra- and Inter-Procedural Modes. RedCard processes methods in one of two modes: an *intra-procedural mode* in which RedCard assumes the most conservative specification "`requires` { } `ensures` { } `modifies` { All }" for all methods and proceeds to infer redundant checks via an intra-procedural dataflow analysis; or an *inter-procedural mode* in which RedCard uses a context-insensitive inter-procedural dataflow analysis to infer both method specifications and redundant checks. The inter-procedural mode also uses a class hierarchy analysis to reason about method resolution. This mode yields better results when precise specifications are not available, but it is also a more complex and compute-intensive analysis.

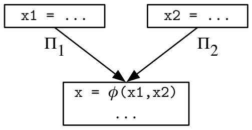
Distinguishing Reads and Writes. Up to this point, our analysis has not distinguished reads and writes. However, it is necessary to do so in practice since precise dynamic race detectors treat them differently. Specifically, two concurrent accesses are considered conflicting only when at least one of them is a write. Given this distinction, RedCard uses the following rules to determine whether a check is redundant:

- A dynamic check on a **write** is redundant if there is a previous write to the same location in the current span.
- A dynamic check on a **read** is redundant if there is a previous read *or write* to the same location in the current span.

Thus, we record in Π whether each access p is a read or a write, and we adjust the definition of redundancy to match the above. For simplicity, we continue to treat all accesses uniformly in the examples below.

Libraries. Since we evaluate RedCard when used in conjunction with FastTrack, RedCard follows FastTrack's treatment of libraries: Fields of library classes are not checked for races, and synchronization operations internal to libraries are assumed to not be used to protect any of the target's data and are ignored. However, several key library methods from `java.lang.Object` and `java.lang.Thread`, such as `Object.notify` and `Thread.start`, are treated specially as synchronizing operations. These assumptions may cause FastTrack to report false alarms (since necessary synchronization within libraries may be ignored), but we have never observed false alarms for the benchmarks studied. RedCard treats synchronization in libraries in exactly the same way, thereby leaving the completeness/soundness guarantees of FastTrack unchanged. The programmer can also provide more precise library module specifications or fully analyze specific library classes via RedCard command-line options.

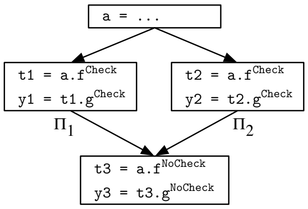
ϕ-functions. WALA's SSA representation makes some aspects of the dataflow analysis much easier since local variables are immutable. However, SSA does have ϕ-functions for merging multiple definitions into a single variable at meet-points in the CFG, as in the following example, which also shows how we compute Π for the entry to the block containing a ϕ-function:



$$\begin{aligned} \Pi &= (\Pi_1[\mathtt{x} := w_1] \cup \{\mathtt{x} = \mathtt{x1}\}) \\ &\sqcap\ (\Pi_2[\mathtt{x} := w_2] \cup \{\mathtt{x} = \mathtt{x2}\}) \end{aligned}$$

In essence, we equate `x` with the appropriate variable in the incoming contexts Π_1 and Π_2. Contexts propagated along back edges introduce one additional complexity. For example, if Π_2 is the context propagated back to the top of a loop containing the definition of `x`, then Π_2 may already include references to `x`. Thus, we replace all occurrences of `x` in Π_2 with a fresh, Skolem variable w_2, and similarly modify Π_1, prior to adding the equalities and computing the meet of the resulting contexts.

Path Expansion. Our previously defined meet operator can be overly coarse at meet-points. To illustrate why, consider the following program and contexts:



$$\Pi_1 = \{\ \mathtt{a.f},\ \mathtt{t1} = \mathtt{a.f},\ \mathtt{t1.g},\ \mathtt{y1} = \mathtt{t1.g}\ \}$$

$$\Pi_2 = \{\ \mathtt{a.f},\ \mathtt{t2} = \mathtt{a.f},\ \mathtt{t2.g},\ \mathtt{y2} = \mathtt{t2.g}\ \}$$

$$\begin{aligned} \Pi_1 \sqcap \Pi_2 &= \{\ \pi \in \Pi_1 \cup \Pi_2 \ \mid\ \Pi_1 \vdash \pi \text{ and } \Pi_2 \vdash \pi\ \} \\ &= \{\mathtt{a.f}\} \end{aligned}$$

Both branches access `a.f.g`. Thus, the check on `y3 = t3.g` is redundant. However, we cannot prove that because the context for the start of the final block would be $\Pi_1 \sqcap \Pi_2 = \{\mathtt{a.f}\}$. In essence, the meet operation loses information about available paths because incoming contexts may encode aliasing via different local variables. To reason more precisely about such situations at meet-points, we permit paths to include more than one field or array reference, and RedCard expands each access path in Π_1 and Π_2, using their respective aliasing constraints, until the access paths refer only to variables defined by instructions common to both control-flow paths leading to the meet point (*i.e.*, that dominate the meet-point). In Π_1 above, `t1.g` and $\mathtt{t1} = \mathtt{a.f}$ are combined to obtain `a.f.g`. The same path is similarly derived from Π_2. These expanded paths are added back into Π_1 and Π_2 before $\sqcap$ is applied, yielding

$$(\Pi_1 \cup \{\mathtt{a.f.g}\}) \sqcap (\Pi_2 \cup \{\mathtt{a.f.g}\}) = \{\ \mathtt{a.f},\ \mathtt{a.f.g}\ \}$$

which does allow us to conclude that checking for races on `y3 = t3.g` is redundant. Supporting longer paths requires changes to PathSet and ModifiesSet operations, but it is mostly straightforward. Our implementation limits paths to contain at most four field or array references to ensure termination.

Loop Invariants. RedCard infers loop invariants with a specialized form of Cartesian predicate abstraction [20, 22] on both access paths and constraints. More specifically, we compute the loop invariant context Π_{inv} in rule [WHILE] by first heuristically generating a conjectured context $\Pi_{heuristic}$ and then repeatedly analyzing the loop body to infer the maximal $\Pi_{inv} \subseteq \Pi_{heuristic}$ that is a valid loop invariant. Our heuristics conjecture invariants based, in part, on the context that first flows to the loop head, inferred loop induction variables, and pattern matching for common idioms. Unusual array access patterns, complex index computations, and irreducible flow graphs may be problematic for our approach, but we found it to work quite well in practice for most loops, including nested loops iterating over multi-dimensional arrays in various ways.

Shadow Proxies. After computing the contexts for every program point, RedCard infers field shadow proxies by conjecturing that each field is a proxy for every other field and then refuting the conjectures that do not hold.

To identify array shadow proxies, RedCard first conjectures a set of possible index relations that hold for all arrays allocated at each allocation site l. These include relations of the form "$\forall i.\ F(i) \succ_l i$" where $F(i)$ is one of the following:

$$\begin{array}{lll} F(i) = 0 & F(i) = (i \text{ div } 4) * 4 & F(i) = i \bmod 4 \\ F(i) = 1 & F(i) = (i \text{ div } 8) * 8 & F(i) = i \bmod 8 \end{array}$$

The first column characterizes whole arrays proxied by a single index, the middle column characterizes arrays divided into chunks where an index is proxied by the first index in its chunk, and the right column characterizes arrays that are always traversed by a fixed stride (*e.g.*, every fourth element is touched). RedCard then refutes the proxy relations for arrays that do not hold. We believe further improvements could be made by expanding this set of conjectures and refining our analysis to reason about more sophisticated patterns.

Code Transformations. A variety of code transformations significantly improve the number of statically identifiable redundant checks. One particularly useful transformation is loop unrolling [8,38]. For example, the following loop (on the left) performs N run-time race checks on `a.f`. Assuming `m` does not modify `a.f`, all of these checks are redundant except the first. Thus the semantically-equivalent version on the right performs only one check.

```
for (i = 0; i < N; i++) {
  a.f^Check.m();
}
```

```
i = 0;
if (i < N) {
  a.f^Check.m();
  for (i = 1; i < N; i++) { a.f^NoCheck.m(); }
}
```

RedCard currently implements loop unrolling via a source-to-source translation step occurring prior to the dataflow analysis. Another transformation we hope to explore in the future is method specialization based on available paths at different call sites. This optimization is synergistic with loop unrolling, since a method called inside the loop body can be specialized to two versions: one for the first iteration outside the loop, and one version for inside the loop where fewer checks may be necessary.

7 Evaluation

We demonstrate the effectiveness of RedCard by evaluating its ability to eliminate redundant checks in a variety of benchmarks. We first describe our experimental framework and analysis time, then show the percentage of access checks removed by RedCard for each program, and finally demonstrate how eliminating checks improves the performance of the FastTrack race detector.

Benchmark Configuration. We performed experiments on the following benchmarks: `elevator`, a discrete event simulator for elevators [38]; `hedc`, a tool to access astrophysics data from Web sources [38]; `tsp`, a Traveling Salesman Problem solver [38]; `mtrt`, a multithreaded ray-tracing program from the SPEC JVM98 benchmark suite [36]; `jbb`, the SPEC JBB2000 business object simulator [36]; `crypt`, `lufact`, `sparse`, `series`, `sor`, `moldyn`, `montecarlo`, and `raytracer` from the Java Grande benchmark suite [25]; the `colt` scientific computing library [6]; the `raja` ray tracer [21]; and and `philo`, a dining philosophers simulation [14]. We configured the Java Grande benchmarks to use four worker threads and the largest data set provided. Table 1 shows statistics for both the "Original" programs and "Unrolled" variants in which RedCard transformed all inner-most loops as described in Section 6. All experiments were performed on an Apple Mac Pro with dual 3GHz quad-core Pentium Xeon processors and 12GB of memory running Sun's Java HotSpot 64-bit Server VM version 1.6.0. Access counts and running times are the average of ten executions.

RedCard Static Analysis Time. The time required by RedCard to process the "Original" benchmarks was on average 18 seconds per thousand lines of code. Approximately 40% of this time was spent loading class files and building the internal data structures used by the WALA analysis engine. The remaining time was primarily consumed by the RedCard dataflow analysis, which has not been optimized. We believe substantial speed ups could be achieved by refactoring and refining our core data structures. Typically, less than 15% of the total running time was spent solving Z3 queries, indicating that the SMT solver was not a bottleneck. The more complex control flow graphs in the "Unrolled" versions led to an average processing time of 24 seconds per thousand lines of code.

Analysis time increased by roughly 50% when RedCard was configured to identify field and array proxies. Our prototype conjectures a large set of possible field and array proxies, and then checks the validity of each conjecture independently. Replacing this approach by a more efficient algorithm that processes multiple conjectures simultaneously would eliminate much of this overhead.

			% Accesses Checked					
			Original			Unrolled		
Program	Size (lines)	Accesses (Millions)	Dynamically NonRedundant	RedCard	RC+Proxy	Dynamically NonRedundant	RedCard	RC+Proxy
colt	111,162	1,102.42	99.6	99.9	99.6	99.3	99.9	99.9
crypt	1,255	2,150.0	56.4	59.2	46.7	40.7	44.6	43.5
lufact	1,627	7,531.07	99.1	99.4	99.4	99.1	99.3	99.3
moldyn	1,409	30,586.08	53.1	59.5	27.5	26.3	27.7	14.8
montecarlo	3,669	2,713.35	2.3	43.0	34.8	2.3	28.8	24.6
mtrt	11,315	24.23	40.2	77.8	77.0	40.2	77.7	76.8
raja	12,027	5.39	55.6	96.8	69.9	55.6	96.8	69.9
raytracer	1,971	39,560.09	51.5	86.2	34.0	48.4	83.2	31.0
sparse	868	7,244.5	83.0	90.5	56.1	34.6	42.1	42.1
series	967	4.0	75.0	83.3	66.7	0.1	33.3	33.3
sor	1,017	2,417.42	83.3	83.5	83.5	66.3	82.8	82.8
tsp	706	820.71	62.9	92.8	86.5	61.3	75.6	69.5
elevator	2,840	0.02	69.2	80.0	66.5	58.5	69.5	61.2
philo	86	<0.01	64.5	67.3	59.4	54.5	59.0	52.2
hedc	24,932	0.05	5.9	94.6	94.5	2.3	93.5	93.4
jbb	45,943	1,068.74	75.8	84.2	78.3	68.4	75.4	69.3
Geo. Mean			46.7	79.3	63.1	24.6	62.6	53.4

Table 1. Percentage of run-time race checks eliminated under different configurations.

Although we have not optimized RedCard for speed, the initial results show that it is no more expensive than existing whole-program techniques. For example, the geometric mean of the time required to process each program in the JavaGrande suite was 101 sec. for Chord and 55 sec. for RedCard. These tools are built on top of frameworks (Soot / WALA) with different performance characteristics, and a more consistent implementation and analysis of them would be required to draw definitive conclusions.

Redundant Check Elimination. Table 1 shows the number of accesses performed by each program. A precise race detector would need to check 100% of those accesses. That table also shows how many of those checks could be eliminated under three scenarios:

Dynamically NonRedundant: To gauge how well the inherently conservative RedCard analysis performs, we first estimate the "optimal" set of instructions that could be annotated as `NoCheck` by examining run-time access histories. Specifically, we ran each program and identified access instructions that only referenced locations already accessed in the current span. We labeled those instructions as `NoCheck` and all others as `Check`. This may over-estimate the number of removable checks due to coverage limitations but seems a reasonable approximation for general comparisons.

RedCard: This column reports the percentage of run-time accesses corresponding to checks that the RedCard static analysis labeled as `Check`.

RC+Proxy: This column also uses RedCard to label accesses, but reasons about shadow proxies for fields and arrays as well.

We show the number of accesses requiring run-time race checks for these scenarios as a percentage of the total number accesses. For the "Original" programs, RedCard reduced the geometric mean of accesses checked to 79.3% of the total number of accesses. While still higher than the Dynamically NonRedundant estimate of 46.7%, it does remove a substantial amount, and for a number of programs, RedCard was close to the estimated optimal.

Inferring shadow proxies proved particularly useful for a number of programs, as shown in the RC+Proxy column. For example, in `raytracer`, RedCard recognized that the `x`, `y`, and `z` fields of a heavily-used 3-D point class were proxies, leading to a reduction in the number of checks performed from 86% to 34%. Other programs, such as `crypt`, contained large arrays for which proxy relations could be computed. Overall, RedCard with proxy inference roughly cut in half the number of run-time race checks.[3] Switching to the *intra-procedural mode* still provided significant benefit, reducing the checking rate to about 83% for RedCard, and to about 70% for RC+Proxy.

For the "Unrolled" programs, RedCard performed somewhat better and reduced the checking rate to roughly 63% for RedCard and to 53% for RC+Proxy.

Overall, these results are quite promising. We note that there is high variance in how well RedCard performs relative to the estimated optimal. In those cases where RedCard performed poorly, imprecisions in aliasing information, array index computations, or loop invariant conjectures made it unable to verify that commonly exercised instructions could be tagged as `NoCheck`. Improving these items would enable RedCard to reason about more subtle code.

Run-time Performance. We now examine how RedCard improves the run-time performance of the FastTrack precise dynamic race detector [18]. FastTrack loads unmodified Java class files and instruments synchronization and memory operations to generate an event stream. It processes those events as the program executes.FastTrack tracks happens-before orderings between synchronization and memory accesses using an adaptive epoch-based representation.

FastTrack's representation of the happens-before relation is quite time and space efficient compared to other precise detectors, but it must check every access and maintain shadow state for every location. Table 2 shows the base running time of our target programs, as well as the slowdown incurred by FastTrack. When running FastTrack and our other analysis tools, all classes loaded by the benchmark programs were instrumented, except those from the standard Java libraries. The timing measurements include the time to load, instrument, and execute the target program startup time. On average, using FastTrack led to a slowdown of 7.5x. Memory accesses are far more common than synchronization

[3] RC+Proxy may yield better results than Dynamically NonRedundant because that estimate does not take possible proxy relationships into account.

			Slowdown (x Base Time)						Shadow Locations	
			Original				Unrolled			
Program	Thread Count	Base Time (sec)	FastTrack	RedCard	RC+Proxy	$\left(\frac{\text{RC+Proxy}}{\text{FastTrack}}\right)$	RedCard	RC+Proxy	Base Count (x 10^5)	RC+Proxy (% of Base)
colt	11	16.0	1.1	1.1	1.1	(0.96)	1.1	1.0	5.1	90.0
crypt	7	1.2	35.4	35.1	13.1	(0.37)	35.0	20.6	1,262.8	13.8
lufact	4	5.6	8.2	7.7	7.7	(0.94)	8.1	8.2	40.1	99.9
moldyn	4	7.5	10.5	7.9	4.7	(0.45)	12.8	6.8	4.7	56.2
montecarlo	4	5.8	9.1	8.2	8.3	(0.91)	8.2	8.0	1,825.2	100.0
mtrt	5	0.4	11.8	10.9	9.5	(0.81)	10.7	9.6	23.0	97.5
raja	2	0.4	6.8	7.2	6.4	(0.94)	7.1	6.5	9.6	59.0
raytracer	4	5.8	14.2	12.5	6.0	(0.42)	12.0	5.9	2,098.6	43.7
sparse	4	5.9	19.9	19.9	19.6	(0.99)	20.6	20.8	159.7	100.0
series	4	150.1	1.0	1.0	1.0	(1.00)	1.0	1.0	20.0	100.0
sor	4	2.3	6.2	6.1	5.9	(0.94)	6.5	6.7	40.0	100.0
tsp	5	0.6	7.3	7.1	7.1	(0.97)	11.1	10.8	1.6	100.0
elevator*	5	5.0	1.2	1.2	1.2	(1.00)	1.2	1.2	<0.1	86.5
philo*	6	9.3	0.6	0.5	0.6	(1.02)	0.6	0.7	<0.1	77.3
hedc*	6	4.7	1.3	1.3	1.3	(0.99)	1.4	1.3	0.4	100.0
jbb*	5	73.9	1.2	1.2	1.2	(1.00)	1.2	1.2	925.0	82.1
Geo. Mean			7.5	7.1	5.7	(0.76)	7.7	6.4		74.8

Table 2. Performance results. Programs marked with * are not compute-bound and are excluded from the mean slowdowns.

operations in Java programs, and the vast majority of FastTrack's overhead is caused by monitoring field and array accesses.

The fourth column in Table 2 shows the slowdown of FastTrack when used in conjunction with RedCard. That redundancy analysis reduced FastTrack's average slowdown from 7.5x to 7.1x. Including shadow proxy analysis further reduced the average slowdown to 5.7x, thereby eliminating approximately 25% of FastTrack's run-time overhead, as computed as the ratio of the running time of FastTrack configured to use RedCard with proxies and FastTrack's original running time in the parenthesized column. The improvements are not linear in the number of checks removed, due to other factors that can impact overall performance, such as lock contention on FastTrack's internal data structures and memory caching effects.

We had expected that the unrolled variants would exhibit greater performance improvements since RedCard eliminated more checks in them. However, they were actually about 15-20% slower on average. The primary cause of the slowdown appears to be the HotSpot JIT compilation engine, which was less able to optimize the unrolled code's larger methods and more complex control structures. We believe a tighter integration of the transformations, analysis, and JIT compilation strategy would mitigate these factors.

The last two columns in Table 2 evaluate RedCard's impact on the shadow memory maintained by FastTrack. The "Base Count" columns shows the number of distinct memory locations accessed by each program (and hence, the number of shadow locations FastTrack must allocate and maintain). A "shadow proxy-aware" FastTrack can avoid allocating shadows for all proxied memory locations, as described in Section 5. For the target programs, this led to a roughly 25% average reduction in the number of allocated shadows. There is high variance among the benchmarks. Some programs in which no interesting proxy relationships were found show little change in the memory footprint, but others, such as `crypt`, `moldyn` and `raytracer`, show dramatic improvement, especially in how many shadow locations were allocated for tracking accesses to arrays.

Comparison to Chord. We also performed preliminary experiments to compare RedCard's ability to find redundancy to that of Chord, a sound static analysis for identifying accesses that may be involved in a data race [30]. Chord and other sound static analyses can often identify specific data structures or memory references guaranteed to be free of races, they may be used to verify or infer `NoCheck` annotations. Chord uses a collection of complex whole-program analyses to reason about reachability, aliasing, locking, and thread ordering. As such, Chord requires much more global reasoning than RedCard's *span-modular* redundancy analysis, particularly when used in the *intra-procedural mode*.

To gauge how well RedCard is able to reason about redundancy when compared to static analyses like Chord, we checked each benchmark program with Chord and labeled all accesses potentially involved in races as `Check`. All other accesses were labeled `NoCheck`. While the average reduction in checks across all benchmarks was roughly on par with RedCard, Chord's ability to reason about individual programs varied greatly. Less than 1% of the run-time checks were eliminated in some programs that use arrays heavily (`crypt`, `moldyn`, `raytracer`, `sparse`, `series`, `sor`). For other programs, greater than 99% of the run-time checks were eliminated (`lufact` and `mtrt`). We believe this bi-modal behavior is caused by several factors: Chord's handling of arrays is less precise than ours, and it seemed able to recognize that heavily used data structures in `lufact` and `mtrt` are thread-local and thus required no checking. Adding a thread-local analysis to RedCard may allow it to similarly remove many of these checks.

8 Related Work

Precise dynamic data race detectors typically represent the happens-before relation with *vector clocks* (VCs) [28], as in the Djit$^+$ race detector [32]. VCs are expensive to maintain, however. The FastTrack race detector uses an adaptive epoch-based representation to reduce the space and time requirements of VCs [18]. Other optimizations include dynamic escape analyses [8,38] or "accordion" vector clocks that reduce space overheads for programs with short-lived threads [10]. Different representations of the happens-before relation have also been explored [14]. Despite these improvements, the overhead of precise detectors can still be prohibitively high.

Similar notions of redundancy and release-free spans have been used in other settings. For example, the IFRit race detector uses the same insight about lock releases in its notion of interference-free regions [13], which were originally designed to facilitate compiler optimizations for race-free programs [12]. The IFRit race detector monitors execution and reports a data race when multiple concurrently executing interference-free regions access the same variable. IFRit is faster than FastTrack, but is not precise and may miss some races. In contrast, RedCard can identify redundancies for any dynamic race detector, without introducing any false positives of false negatives. That is, any accesses identified as redundant can be skipped by any race detector without changing the guarantees provided by that detector.

Another study uses similar concepts to verify programmer-specified ownership policies [27], but that analysis is more restricted in its ability to reason about aliasing and function calls, and it requires programmers to specify when a thread has exclusive access to memory locations, rather than inferring it. Raman et al. developed a race condition algorithm for the very different context of structured parallelism, as in Cilk or X10 [33]. Their technique applies check hoisting and check elimination optimizations similar in spirit to RedCard to this domain. Their experimental results find fewer opportunities for optimization than RedCard, most likely due to either limitations in the precision in their analysis or the nature of programs written for such systems.

Many static analysis techniques for identifying races have also been explored, including systems based on types [1,3,23], model checking [7,29,42] and dataflow analysis [15], as well as scalable whole-program analyses [30,40]. While static race detection provides the potential to detect all race conditions over all program paths, decidability limitations imply that any sound static race detector may produce false alarms. As mentioned previously, these static analyses can often identify specific data structures or memory references guaranteed to be free of races and may thus be used to verify or infer `NoCheck` annotations. However, soundness of the static analysis is essential to avoid missing data races. Many of the mentioned static analyses are either unsound by design or unsound in their implementations to reduce the number of spurious warnings (see, *e.g.*, [1, 15]). Their focus on identifying race-free accesses rather than redundant race checks also lead to different design choices in terms of precision and scalability.

Gross *et al.* present a global static analysis to improve the precision and performance of a LockSet-based detector [38]. In contrast to RedCard's focus on redundant checks, their analysis, while it does eliminate some redundant checks on field accesses in a fairly restrictive way, is primarily designed to identify objects on which no races can occur. As such, their algorithm requires global aliasing information, as well as a static approximation of the happens-before graph for the whole program. Moreover, their reliance on an imprecise race detector leads their system to both miss races and report spurious warnings. They also do not support arrays.

Choi *et al.* present a different global analysis for removing run-time race checks for accesses guaranteed to be race-free [8]. Despite the primary focus on identifying race-free accesses, it does include elements closer in spirit to RedCard. In particular, the analysis eliminates some redundant checks via a simple intra-procedural analysis. However, their notion of redundancy is less general than ours. They cannot, for example, track redundancy across method calls, reason about array accesses, or model synchronization operations as precisely as RedCard. In addition, their analysis is tailored for use with a variant of the imprecise LockSet algorithm. RedCard can be used to optimize any detector.

Various alias analyses have used notions similar to our available paths. For example, Fink *et al.* compute must and must-not aliases via access paths as part of an analysis to verify type-state protocols [16]. RedCard goes beyond that approach by supporting synchronization operations, a more precise model of arrays, and reasoning about race conditions. At first glance, RedCard seems to perform similar reasoning to analyses for redundant load elimination optimizations in compilers for concurrent languages, as in [4, 24, 39] for example. Our notion of redundancy, however, focuses on which locations have been accessed, and not on whether a value read from memory may be subsequently reused, as illustrated in Section 3. This leads to a more general notion of redundancy more amenable to analysis by tools specifically designed to reason about it.

Acknowledgments. This work was supported by NSF grants 0905650, 1116883 and 1116825. We thank James Wilcox for his assistance on the experiments.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2):207–255, 2006.
2. S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA*, pages 234–243, 1991.
3. R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, 2004.
4. R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT*, pages 41–52, 2009.
5. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, 2001.
6. CERN. Colt 1.2.0. `http://dsd.lbl.gov/~hoschek/colt/`.
7. A. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst, 1996.
8. J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, 2002.
9. J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *TOPLAS*, 13(4):491–530, 1991.
10. M. Christiaens and K. D. Bosschere. TRaDe: Data Race Detection for Java. In *International Conference on Computational Science*, pages 761–770, 2001.
11. L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

12. L. Effinger-Dean, H.-J. Boehm, D. R. Chakrabarti, and P. G. Joisha. Extended sequential reasoning for data-race-free programs. In *MSPC*, pages 22–29, 2011.
13. L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: interference-free regions for dynamic data-race detection. In *OOPSLA*, pages 467–484, 2012.
14. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *PLDI*, pages 245–255, 2007.
15. D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.
16. S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *TOSEM*, 17(2), 2008.
17. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, pages 219–232, 2000.
18. C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.
19. C. Flanagan and S. N. Freund. Redcard: Redundant check elimination for dynamic race detectors. Technical Report UCSC-SOE-13-05, UC Santa Cruz, 2013.
20. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
21. E. Fleury and G. Sutre. Raja, version 0.4.0-pre4. Available at `http://raja.sourceforge.net/`, 2007.
22. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
23. D. Grossman. Type-safe multithreading in Cyclone. In *TLDI*, 2003.
24. A. L. Hosking, N. Nystrom, D. Whitlock, Q. I. Cutts, and A. Diwan. Partial redundancy elimination for access path expressions. *Softw., Pract. Exper.*, 31(6):577–600, 2001.
25. Java Grande Forum. Java Grande benchmark suite. Available at `http://www.javagrande.org`, 2008.
26. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
27. J.-P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro. Dynamically checking ownership policies in concurrent C/C++ programs. In *POPL*, pages 457–470, 2010.
28. F. Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, 1989.
29. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
30. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.
31. H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.
32. E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.
33. R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *RV*, pages 368–383, 2010.
34. M. Ronsse and K. D. Bosschere. RecPlay: A fully integrated practical record/replay system. *TCS*, 17(2):133–152, 1999.

35. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *TOCS*, 15(4):391–411, 1997.
36. Standard Performance Evaluation Corporation. SPEC benchmarks. `http://www.spec.org/`, 2003.
37. C. von Praun and T. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.
38. C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI*, pages 115–128, 2003.
39. C. von Praun, F. T. Schneider, and T. R. Gross. Load elimination in the presence of side effects, concurrency and precise exceptions. In *LCPC*, pages 390–405, 2003.
40. J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *FSE*, pages 205–214, 2007.
41. T.J. Watson Libraries for Analysis (WALA). Available at `http://wala.sourceforge.net/`, 2012.
42. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL*, pages 27–40, 2001.
43. Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, pages 221–234, 2005.

A Formal Development (Supplemental Material)

This Appendix presents supplementary details of the RedJava semantics, type system, and main correctness theorems presented in Sections 2 and 3. Arrays, quantified access paths, and proxies are omitted from the development for simplicity. They pose no major technical challenge and are relatively straight-forward to include, but they do introduce additional notational complexity.

A.1 Semantics

We specify the operational semantics of RedJava using the abstract machine in Figure 6. The machine evaluates a program by stepping through a sequence of states. Each state Σ consists of two components: a heap H and a collection of threads T. The heap maps locations to values, where each location $\rho.f$ combines an object address ρ with a field name f. The heap also maps each object address ρ to the thread identifier (or Tid) of the thread holding the object's lock (or $\perp$ if it is not held). The thread set T maps each thread identifier $t \in \mathit{Tid}$ to a thread state $\langle \sigma, s \rangle$ that combines a statement s with a (thread-local) store σ mapping variables in s to values.

In the context of a set of definitions $\overline{D}$, the relation

$$\overline{D} \vdash H \bullet \langle \sigma, s \rangle \longrightarrow^{a} H' \bullet \langle \sigma', s' \rangle$$

models the effect of a single step by thread $\langle \sigma, s \rangle$ on the heap H and the thread's local state. The *Action* a captures the heap operation performed by the step. For example, if thread t accesses location $\rho.f^k$, a would be $t : \mathtt{acc}(\rho.f^k)$. The special action $t{:}\epsilon$ indicates that a step has no heap effect.

Figure 6 defines the evaluation rules for each statement form. In these rules, evaluation context $\mathcal{E}$ captures the sequential execution of statements within a thread, and the heap $H[\rho.f := v]$ is identical to H except that it maps the location $\rho.f$ to the value v. Similar update operations are used on the other state components. For example, $H[\rho := t]$ updates H to indicate that the lock for the object at location ρ is held by t. The term $\sigma(e)$ evaluates an expression e using local store σ for the values of variables.

To invoke a method x = $y.m(\overline{z})$, we first look up the method m in the program definitions. (For simplicity, we do not model dynamic lookup in the formal semantics.) We then construct a substitution θ that maps 1) m's local variables, which are the free variables of s, other than the return result variable r, to fresh names, 2) the parameters $\overline{z'}$ to the arguments $\overline{z}$, 3) the self-reference `this` to y, and 4) the return variable r to x. If s is the method body of m, $\theta(s)$ may be inserted into the evaluation context surrounding the call without variable capture. Moreover, the result of the call is placed in x, as expected.

The relation $\overline{D} \vdash \Sigma \longrightarrow^{a} \Sigma'$ describes a single step of multithreaded program execution. That rule selects an arbitrary thread t to take a step and updates the global state Σ accordingly. As above, a captures the memory or synchronization operation performed by the step. We use the notation $t{:}\_$ to represent an arbitrary action by thread t.

Figure 6: Run-time States and Transition Relation

States

$$\begin{array}{rcl}
\Sigma \in \mathit{State} & ::= & H\bullet T \\
H \in \mathit{Heap} & = & (\mathit{Location} \mapsto \mathit{Value}) \cup (\mathit{Address} \mapsto \mathit{Tid}_\perp) \\
l \in \mathit{Location} & ::= & \rho.f \\
u,t \in \mathit{Tid} & ::= & 1 \mid 2 \mid \ \ldots \\
T \in \mathit{Threads} & = & \mathit{Tid} \mapsto \langle \sigma, s \rangle \\
\sigma \in \mathit{Store} & = & \mathit{Var} \mapsto \mathit{Value} \\
\mathcal{E} \in \mathit{EvalCtxt} & ::= & [\,] \ \mid \ \mathcal{E}; s
\end{array}$$

Traces

$$\begin{array}{rcl}
\alpha \in \mathit{Trace} & ::= & \overline{\mathit{Action}} \\
a, b \in \mathit{Action} & ::= & t : \mathtt{acc}(\rho.f^k) \\
 & \mid & t : \mathtt{acq}(\rho) \\
 & \mid & t : \mathtt{rel}(\rho) \\
 & \mid & t : \epsilon
\end{array}$$

$$\boxed{\overline{D} \vdash \Sigma \longrightarrow^a \Sigma'}$$

$$\overline{D} \vdash H\bullet T[t := \langle \sigma, s \rangle] \quad \longrightarrow^a \quad H\bullet T[t := \langle \sigma', s' \rangle] \qquad \text{if } \overline{D} \vdash H\bullet\langle \sigma, s \rangle \longrightarrow^a H'\bullet\langle \sigma', s' \rangle \text{ and } a = t : \_$$

$$\boxed{\overline{D} \vdash H\bullet\langle \sigma, s \rangle \longrightarrow^a H'\bullet\langle \sigma', s' \rangle}$$

$$\begin{array}{llll}
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[\mathtt{skip}; s] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma, \mathcal{E}[s] \rangle & \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[\mathtt{if}\ \mathit{be}\ s_1\ s_2] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma, \mathcal{E}[s_1] \rangle & \text{if } \sigma(e) = \mathtt{true} \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[\mathtt{if}\ \mathit{be}\ s_1\ s_2] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma, \mathcal{E}[s_2] \rangle & \text{if } \sigma(e) = \mathtt{false} \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[\mathtt{while}\ \mathit{be}\ s] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma, \mathcal{E}[\mathtt{if}\ \mathit{be}\ (s; \mathtt{while}\ \mathit{be}\ s)\ \mathtt{skip}] \rangle & \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[x\ \mathtt{=}\ y.f^k] \rangle & \longrightarrow^{t:\mathtt{acc}(\rho.f^k)} & H\bullet\langle \sigma[x := v], \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \sigma(y) = \rho \text{ and } H(\rho.f) = v \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[y.f^k\ \mathtt{=}\ x] \rangle & \longrightarrow^{t:\mathtt{acc}(\rho.f^k)} & H[\rho.f := v]\bullet\langle \sigma, \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \sigma(y) = \rho \text{ and } \sigma(x) = v \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[\mathtt{acq}\ x] \rangle & \longrightarrow^{t:\mathtt{acq}(\rho)} & H[\rho := t]\bullet\langle \sigma, \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \sigma(x) = \rho \text{ and } H(\rho) = \perp \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[\mathtt{rel}\ x] \rangle & \longrightarrow^{t:\mathtt{rel}(\rho)} & H[\rho := \perp]\bullet\langle \sigma, \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \sigma(x) = \rho \text{ and } H(\rho) = t \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[x\ \mathtt{=}\ \mathtt{new}\ C] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma[x := \rho], \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \rho \text{ is fresh} \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[x\ \mathtt{=}\ e] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma[x := v], \mathcal{E}[\mathtt{skip}] \rangle & \text{if } \sigma(e) = v \\
\overline{D} \vdash H\bullet\langle \sigma, \mathcal{E}[x\ \mathtt{=}\ y.m\mathtt{(}\overline{z}\mathtt{)}] \rangle & \longrightarrow^{t:\epsilon} & H\bullet\langle \sigma, \mathcal{E}[\theta(s)] \rangle & \\
 & & \quad \text{if } m\mathtt{(}\overline{z'}\mathtt{)}\ \mathit{spec}\ \{\ s; \mathtt{return}\ r\ \} \in \overline{D} & \\
 & & \quad\ \text{and } \theta \text{ maps } \mathrm{FV}(s) \setminus \{r\} \text{ to fresh names} & \\
 & & \quad\ \text{and } \theta \text{ maps } \overline{z'}, \mathtt{this}, r \text{ to } \overline{z}, y, x, \text{ respectively.} &
\end{array}$$

The relation $\longrightarrow^{\alpha}$ denotes the reflexive-transitive closure of $\longrightarrow^{a}$, where the *trace* α is a sequence of actions $a_1 \cdot a_2 \cdots a_n$. Given this definition, $\overline{D} \vdash \Sigma_0 \longrightarrow^{\alpha} \Sigma$ models the arbitrary interleaving of the various threads of a multithreaded program $\overline{D}$, starting from the initial execution state Σ_0. For a program $\overline{D}\ s_1 \| \ldots \| s_n$, this initial execution state is $\Sigma_0 = H_0 \bullet T_0$, where

- H_0 maps all locations to `null` and all addresses to $\bot$; and
- T_0 maps each thread $t \in 1..n$ to $\langle \sigma, s_t \rangle$, where σ assigns a distinct global address to each free variable in $s_{1..n}$. Thus, free variables in $s_{1..n}$ implicitly denote potentially thread-shared objects.

A.2 Race Conditions and Dynamic Race Detection

As mentioned above, the happens-before relation for trace α is the smallest reflexive, transitive relation $<_{\alpha}$ on α such that $a <_{\alpha} b$ if a occurs before b in α and either:

1. a and b are performed by the same thread; or
2. a releases some lock and b acquires that lock.

Any well-formed trace has a race condition if and only if it has a detected race condition.

Restatement of Theorem 1 (Race Detection). *Suppose α is a well-formed trace. Then α has a race condition iff α has a detected race condition.*

Proof. We show that if α has a data race, then it has a detected data race. (The other direction is trivial.) Consider any race in α between conflicting accesses a and b to some location $\rho.f$ by distinct threads t_a and t_b. Since α is well-formed, a must be preceded by (or equal to) a checked access $a' = t_a : \mathtt{acc}(\rho.f^{\mathtt{Check}})$ with no intervening release by t_a. A similar such $b' = t_b : \mathtt{acc}(\rho.f^{\mathtt{Check}})$ exists for b.

Suppose a' and b' do not race. Then

$$a' <_{\alpha} b'$$

which means that, for some lock m,

$$a' \;<_{\alpha}\; t_a{:}\mathtt{rel}(m) \;<_{\alpha}\; b'$$

Moreover,

$$a' \;<_{\alpha}\; a \;<_{\alpha}\; t_a{:}\mathtt{rel}(m) \;<_{\alpha}\; b' \;<_{\alpha}\; b$$

because $a' <_{\alpha} a$ (with no intervening releases) and $b' <_{\alpha} b$. Thus, there is no race between a and b and we reach a contradiction. Therefore, a' and b' do cause a race condition, and it is a checked race condition since both actions are marked as `Check`. □

Figure 7: Type Rules for Run-time States.

$\boxed{\overline{D}; \alpha \vdash \Sigma}$

[State]

$$\frac{\begin{array}{c} \vdash \overline{D} \\ \overline{D}; \alpha \vdash_t H \bullet T(t) \qquad \forall t \in \mathit{Tid} \end{array}}{\overline{D}; \alpha \vdash H \bullet T}$$

$\boxed{\overline{D}; \alpha \vdash_t H \bullet \langle \sigma, s \rangle}$

[Thread]

$$\frac{\begin{array}{c} \sigma' \text{ extends } \sigma \text{ with a mapping for the} \\ \text{Skolem variables in } \Pi \\ H; \sigma'; \alpha \vdash_t \Pi \\ \overline{D} \vdash s : \Pi \leadsto \Pi' \end{array}}{\overline{D}; \alpha \vdash_t H \bullet \langle \sigma, s \rangle}$$

$\boxed{H; \sigma; \alpha \vdash_t \Pi}$

[Context]

$$\frac{H; \sigma; \alpha \vdash_t \pi \qquad \forall \pi \in \Pi}{H; \sigma; \alpha \vdash_t \Pi}$$

$\boxed{H; \sigma; \alpha \vdash_t \pi}$

[Path]

$$\frac{\begin{array}{c} \alpha = \alpha_1 \cdot t\!:\!\mathtt{acc}(\sigma(y).f^{\mathtt{Check}}) \cdot \alpha_2 \\ \alpha_2 \text{ contains no releases by } t \end{array}}{H; \sigma; \alpha \vdash_t y.f}$$

[Alias Constraint]

$$\frac{\begin{array}{c} H; \sigma; \alpha \vdash_t y.f \\ \sigma(x) = H(\sigma(y.f)) \end{array}}{H; \sigma; \alpha \vdash_t x \ \mathtt{=}\ y.f}$$

[Boolean Constraint]

$$\frac{\sigma(be) = \mathtt{true}}{H; \sigma; \alpha \vdash_t be}$$

A.3 Type System

Run-Time States. We extend our type system to run-time states Σ via the judgment $\overline{D}; \alpha \vdash \Sigma$ defined in Figure 7. Here, the trace α provides information about previous accesses within current spans.

The related judgment $\overline{D}; \alpha \vdash_t H \bullet \langle \sigma, s \rangle$ checks a particular thread $\langle \sigma, s \rangle$ in the context of heap H and trace α. The context Π may refer to Skolem variables, so this rule first extends the local store σ to a store σ' containing arbitrary values for those variables. It then uses the judgment $H; \sigma; \alpha \vdash_t \Pi$ to check if the available paths and constraints in Π are satisfied by heap H, local store σ', and history trace α.

Context Implication. As described in Section 3.1, the RedJava type system relies upon the context implication relation $\Pi \vdash \pi$ for verifying indicate that a path, alias constraint, or equality constraint π is satisfied by the context Π. Any decision procedure may be used to implement this relation, provided that it is sound. The following property formalizes this soundness assumption, essentially requiring that any context Π' implied by Π is true in any run-time state conforming to Π.

Property 1 (Sound Context Implication). *Suppose* $H; \sigma; \alpha \vdash_t \Pi$. *If* $\Pi \sqsubseteq \Pi'$ *then* $H; \sigma; \alpha \vdash_t \Pi'$.

In addition, we require that the implication decision procedure be transitive.

Property 2 (Transitive Context Implication). *If $\Pi \sqsubseteq \Pi'$ and $\Pi' \sqsubseteq \Pi''$ then $\Pi \sqsubseteq \Pi''$.*

A.4 Correctness

We now address the correctness guarantee provided by our analysis. Our first main correctness result is a preservation theorem showing that if our analysis can verify the `NoCheck` annotations in a run-time state Σ, then it can also verify any successor state Σ' with respect to an appropriately extended trace, provided that trace is race-free.

Restatement of Theorem 2 (Preservation). *If $\overline{D}; \alpha \vdash \Sigma$ and $\Sigma \longrightarrow^a \Sigma'$ and $\alpha \cdot a$ is race-free, then $D; (\alpha \cdot a) \vdash \Sigma'$.*

Proof. Suppose the action a is performed by thread t. From rule [State] and the definition of our transition relation, we have the following:

$$\begin{array}{l}
\vdash \overline{D} \\
\overline{D}; \alpha \vdash_i H \bullet T(i) \qquad \forall i \in \mathit{Tid} \\
\Sigma = H \bullet T[t := \langle \sigma, s \rangle] \\
\Sigma' = H' \bullet T[t := \langle \sigma', s' \rangle] \\
\overline{D} \vdash H \bullet \langle \sigma, s \rangle \longrightarrow^a H' \bullet \langle \sigma', s' \rangle
\end{array}$$

In addition, for any thread $T(i) = \langle \sigma_i, s_i \rangle$, rule [Thread] requires that there is an extension σ'_i of σ_i such that:

$$\begin{array}{l}
H; \sigma'_i; \alpha \vdash_i \Pi_i \\
\overline{D} \vdash s_i : \Pi_i \leadsto \Pi'_i
\end{array}$$

for some Π_i and Π'_i. We now show that $\overline{D}; (\alpha \cdot a) \vdash H' \bullet T(i)$ for all $i \neq t$.

- If $a = t\!:\!\epsilon$ or $a = t\!:\!\texttt{rd}(\rho.f)$, then $H' = H$ and this follows immediately.
- If $a = t : \texttt{wr}(\rho.f)$, then H and H' may differ at $\rho.f$. However, if $\alpha \cdot a$ is race-free, then any access to $\rho.f$ in α by thread i is followed by a release by the same thread. Neither $y.f$ or $x = y.f$ will be in Π_i for any y such that $\sigma'_i(y) = \rho$. So, $H'; \sigma'_i; (\alpha \cdot a) \vdash \Pi_i$. Thus, $\overline{D}; (\alpha \cdot a) \vdash H' \bullet T(i)$ for all $i \neq t$.
- If $a = t\!:\!\texttt{acq}(\rho)$ or $a = t\!:\!\texttt{rel}(\rho)$, then heaps H' and H agree on the values stored at all locations. Items in Π_i may refer to variables mapping to heap locations but never to the status of the lock $H(\rho)$. Thus, it follows that $H'; \sigma'_i; (\alpha \cdot a) \vdash_i \Pi_i$ for all $i \neq t$.

By Lemma 1 below, $\overline{D}; (\alpha \cdot a) \vdash_t H' \bullet \langle \sigma', s' \rangle$. Hence, $\overline{D}; (\alpha \cdot a) \vdash \Sigma'$ then follows by rule [State]. □

We now show that any step taken by a thread from a well-formed thread state leads to another well-formed state.

Lemma 1 (Thread-Step Preservation). *Suppose* $\vdash \overline{D}$ *and* a *is an action by thread* t*. If* $\overline{D}; \alpha \vdash_t H \bullet \langle \sigma, s \rangle$ *and* $\overline{D} \vdash H \bullet \langle \sigma, s \rangle \longrightarrow^a H' \bullet \langle \sigma', s' \rangle$*, then* $\overline{D}; (\alpha \cdot a) \vdash_t H' \bullet \langle \sigma', s' \rangle$*.*

Proof. Since $\overline{D}; \alpha \vdash_t H \bullet \langle \sigma, s \rangle$, there exists Π_1, Π_2, and σ'' containing Skolem bindings such that:

$$\begin{array}{l} H; (\sigma \uplus \sigma''); \alpha \vdash_t \Pi_1 \\ \overline{D} \vdash s : \Pi_1 \rightsquigarrow \Pi_2 \end{array}$$

Given that $\overline{D} \vdash H \bullet \langle \sigma, s \rangle \longrightarrow^a H' \bullet \langle \sigma', s' \rangle$, we also have that

$$\overline{D} \vdash H \bullet \langle (\sigma \uplus \sigma''), s \rangle \longrightarrow^a H' \bullet \langle (\sigma' \uplus \sigma''), s' \rangle$$

since the extra Skolem bindings do not influence evaluation. By Lemma 2 (below), there exist Π_3 and Π_4 and σ''' containing bindings for fresh Skolem variables such that:

$$\begin{array}{l} \overline{D} \vdash s' : \Pi_3 \rightsquigarrow \Pi_4 \\ \Pi_4 \sqsubseteq \Pi_2 \\ H'; (\sigma' \uplus \sigma'' \uplus \sigma'''); (\alpha \cdot a) \vdash_t \Pi_3 \end{array}$$

and hence $\overline{D}; (\alpha \cdot a) \vdash_t H' \bullet \langle \sigma', s' \rangle$. □

Lemma 2. *If* $\vdash \overline{D}$ *and* a *is an action by thread* t *and*

$$\begin{array}{l} \overline{D} \vdash s : \Pi_1 \rightsquigarrow \Pi_2 \\ H; \sigma; \alpha \vdash_t \Pi_1 \\ \overline{D} \vdash H \bullet \langle \sigma, s \rangle \longrightarrow^a H' \bullet \langle \sigma', s' \rangle \end{array}$$

then there exist Π_3 *and* Π_4 *and* σ'' *containing bindings for fresh Skolem variables such that:*

$$\begin{array}{l} \overline{D} \vdash s' : \Pi_3 \rightsquigarrow \Pi_4 \\ \Pi_4 \sqsubseteq \Pi_2 \\ H'; (\sigma' \uplus \sigma''); (\alpha \cdot a) \vdash_t \Pi_3 \end{array}$$

Proof. Proof is by induction on the structure of s.

- $s = s_1; s_2$ where $s_1 \neq \texttt{skip}$: In this case, $s' = s_1'; s_2$ where

  $$\begin{array}{l} \overline{D} \vdash s_1 : \Pi_1 \rightsquigarrow \Pi_5 \\ \overline{D} \vdash s_2 : \Pi_5 \rightsquigarrow \Pi_2 \\ \overline{D} \vdash H \bullet \langle \sigma, s_1 \rangle \longrightarrow^a H' \bullet \langle \sigma', s_1' \rangle \end{array}$$

  By the inductive hypothesis, there exist Π_3 and Π_6 and σ'' such that

  $$\begin{array}{l} \overline{D} \vdash s_1' : \Pi_3 \rightsquigarrow \Pi_6 \\ \Pi_6 \sqsubseteq \Pi_5 \\ H'; (\sigma' \uplus \sigma''); (\alpha \cdot a) \vdash_t \Pi_3 \end{array}$$

  By Lemma 5, there exists Π_4 such that $\overline{D} \vdash s_2 : \Pi_6 \rightsquigarrow \Pi_4$ with $\Pi_4 \sqsubseteq \Pi_2$. Thus, $\overline{D} \vdash s' : \Pi_3 \rightsquigarrow \Pi_4$ by [Seq].

- $s = \texttt{skip}; s'$: In this case

$$
\begin{array}{l}
H' = H \\
\sigma' = \sigma \\
a = t\!:\!\epsilon \\
\overline{D} \vdash \texttt{skip} : \Pi_1 \leadsto \Pi_1 \\
\overline{D} \vdash s' : \Pi_1 \leadsto \Pi_2 \\
H; \sigma; \alpha \vdash_t \Pi_1
\end{array}
$$

Let $\Pi_3 = \Pi_1$ and $\Pi_4 = \Pi_2$ and $\sigma'' = \emptyset$ (the empty store). Then:

$$
\begin{array}{ll}
\overline{D} \vdash s' : \Pi_3 \leadsto \Pi_4 & \text{since } \Pi_3 = \Pi_1 \\
\Pi_4 \sqsubseteq \Pi_2 & \text{since } \Pi_4 = \Pi_2 \\
H; \sigma; (\alpha \cdot t\!:\!\epsilon) \vdash_t \Pi_3 & \text{via Lemma 3 and } \Pi_3 = \Pi_1
\end{array}
$$

- $s = \texttt{if}\ \mathit{be}\ s_1\ s_2$: Suppose $\sigma(\mathit{be}) = \texttt{true}$. (The case when it is false is similar.)

$$
\begin{array}{l}
s' = s_1 \\
H' = H \\
\sigma' = \sigma \\
a = t\!:\!\epsilon \\
\overline{D} \vdash s_1 : \Pi_1 \cup \{\mathit{be}\} \leadsto \Pi_1' \\
\overline{D} \vdash s_2 : \Pi_1 \cup \{\neg \mathit{be}\} \leadsto \Pi_2' \\
\Pi_2 = \Pi_1' \sqcap \Pi_2' = \{\ \pi \in \Pi_1' \cup \Pi_2'\ \mid\ \Pi_1' \vdash \pi \text{ and } \Pi_2' \vdash \pi\ \}
\end{array}
$$

Let $\Pi_3 = \Pi_1 \cup \{\mathit{be}\}$ and $\Pi_4 = \Pi_1'$ and $\sigma'' = \emptyset$. Then:

$$
\begin{array}{ll}
\overline{D} \vdash s' : \Pi_3 \leadsto \Pi_4 & \text{since } \Pi_3 = \Pi_1 \cup \{\mathit{be}\} \text{ and } \Pi_4 = \Pi_1' \\
\Pi_4 \sqsubseteq \Pi_2 & \text{since } \forall \pi \in \Pi_2.\ \Pi_1' \vdash \pi \\
H; \sigma; (\alpha \cdot t\!:\!\epsilon) \vdash_t \Pi_3 & \text{via Lemma 4 and Lemma 3}
\end{array}
$$

- $s = \texttt{while}\ \mathit{be}\ s_1$: In this case,

$$
\begin{array}{l}
s' = \texttt{if}\ \mathit{be}\ (s_1; s)\ \texttt{skip} \\
H' = H \\
\sigma' = \sigma \\
a = t\!:\!\epsilon \\
\Pi_1 \sqsubseteq \Pi_{\mathit{inv}} \\
\Pi_2 = \Pi_{\mathit{inv}} \cup \{\neg \mathit{be}\} \\
\overline{D} \vdash s_1 : \Pi_{\mathit{inv}} \cup \{\mathit{be}\} \leadsto \Pi' \\
\Pi' \sqsubseteq \Pi_{\mathit{inv}}
\end{array}
$$

Let $\Pi_3 = \Pi_{\mathit{inv}}$ and $\Pi_4 = \Pi_2$ and $\sigma'' = \emptyset$. Then:

$$
\begin{array}{ll}
\overline{D} \vdash s : \Pi' \leadsto \Pi_4 & \text{via [\textsc{While}]} \\
\overline{D} \vdash s_1; s : \Pi_{\mathit{inv}} \cup \{\mathit{be}\} \leadsto \Pi_4 & \text{via [\textsc{Seq}]} \\
\overline{D} \vdash \texttt{skip} : \Pi_{\mathit{inv}} \cup \{\neg \mathit{be}\} \leadsto \Pi_4 & \text{via [\textsc{Skip}]} \\
\overline{D} \vdash s' : \Pi_3 \leadsto \Pi_4 & \text{via [\textsc{If}]}
\end{array}
$$

Also,

$$\begin{array}{ll} H; \sigma; (\alpha \cdot t\!:\!\epsilon) \vdash_t \mathit{\Pi}_1 & \text{via Lemma 3} \\ H; \sigma; (\alpha \cdot t\!:\!\epsilon) \vdash_t \mathit{\Pi}_3 & \text{via Property 1} \end{array}$$

- $s = \texttt{rel}\ x$: In this case

  $$\begin{array}{l} s' = \texttt{skip} \\ \sigma(x) = \rho \\ H(\rho) = t \\ H' = H[\rho := \bot] \\ \sigma' = \sigma \\ a = t\!:\!\texttt{rel}(x) \\ \mathit{\Pi}_2 = \mathit{\Pi}_1 \setminus \textsc{All} = \{\mathit{be} \in \mathit{\Pi}_1\} \end{array}$$

  Let $\mathit{\Pi}_3 = \mathit{\Pi}_4 = \mathit{\Pi}_2$ and $\sigma'' = \emptyset$. Then $\overline{D} \vdash s' : \mathit{\Pi}_3 \rightsquigarrow \mathit{\Pi}_4$ via [Skip], and $H'; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_3$ holds since $\mathit{\Pi}_3$ contains only boolean expressions that are true in σ.
- $s = \texttt{acq}(x)$: In this case

  $$\begin{array}{l} s' = \texttt{skip} \\ \sigma(x) = \rho \\ H(\rho) = \bot \\ H' = H[\rho := t] \\ \sigma' = \sigma \\ a = t\!:\!\texttt{acq}(x) \\ \mathit{\Pi}_2 = \mathit{\Pi}_1 \end{array}$$

  Let $\mathit{\Pi}_3 = \mathit{\Pi}_4 = \mathit{\Pi}_2$ and $\sigma'' = \emptyset$. Then $\overline{D} \vdash s' : \mathit{\Pi}_3 \rightsquigarrow \mathit{\Pi}_4$ via [Skip]. Also, $H'; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_3$ holds via Lemma 3 and the fact that no item in $\mathit{\Pi}_3$ refers to $H(\rho)$.
- $s = (x\ \texttt{=}\ \texttt{new}\ C)$: In this case

  $$\begin{array}{l} s' = \texttt{skip} \\ H' = H \\ \sigma' = \sigma[x := \rho] \\ \rho \text{ is fresh} \\ a = t\!:\!\epsilon \\ \mathit{\Pi}_2 = \mathit{\Pi}_1[x := w] \end{array}$$

  Let $\mathit{\Pi}_3 = \mathit{\Pi}_4 = \mathit{\Pi}_2$ and $\sigma'' = [w := \sigma(c)]$. Then $\overline{D} \vdash s' : \mathit{\Pi}_3 \rightsquigarrow \mathit{\Pi}_4$ via [Skip].
- $s = (x\ \texttt{=}\ e)$: In this case

  $$\begin{array}{l} s' = \texttt{skip} \\ \sigma(e) = v \\ H' = H \\ \sigma' = \sigma[x := v] \\ a = t\!:\!\epsilon \\ \mathit{\Pi}_2 = \mathit{\Pi}_1[x := w] \cup \{x = e[x := w]\} \end{array}$$

Let $\mathit{\Pi}_3 = \mathit{\Pi}_4 = \mathit{\Pi}_2$ and $\sigma'' = [w := \sigma(x)]$. Then: $\overline{D} \vdash s' : \mathit{\Pi}_3 \leadsto \mathit{\Pi}_4$ via [Skip].

Also, $H'; (\sigma' \uplus \sigma''); \alpha \vdash_t \mathit{\Pi}_3$. Lemma 3 then implies that $H; \sigma'; (\alpha \cdot a) \vdash_t \mathit{\Pi}_3$.

– $s = (x \texttt{ = } y.f^k)$: In this case,

$$\begin{array}{l} s' = \texttt{skip} \\ \sigma(y) = \rho \\ H(\rho.f) = v \\ H' = H \\ \sigma' = \sigma[x := v] \\ a = t\!:\!\texttt{acc}(\rho.f^k) \\ k = \texttt{NoCheck} \Rightarrow \mathit{\Pi}_1 \vdash y.f \\ \mathit{\Pi}_2 = \mathit{\Pi}_1[x := w] \cup \{x = (y.f)[x := w], (y.f)[x := w]\} \end{array}$$

Let $\mathit{\Pi}_3 = \mathit{\Pi}_4 = \mathit{\Pi}_2$ and $\sigma'' = [w := \sigma(x)]$. Rule [Skip] shows that $\overline{D} \vdash s' : \mathit{\Pi}_3 \leadsto \mathit{\Pi}_4$.

Then $H; \sigma' \uplus \sigma''; (\alpha \cdot a) \vdash_t \mathit{\Pi}_3$ follows by Lemma 3. We now consider the two cases for k:

- Suppose $k = \texttt{Check}$. Then $H; (\sigma' \uplus \sigma''); (\alpha \cdot a) \vdash_t (y.f)[x := w]$ since $(\sigma' \uplus \sigma'')((y.f)[x := w]) = \rho.f$ and a is a checked access to $\rho.f$ by t. Thus,

$$H; (\sigma' \uplus \sigma''); (\alpha \cdot a) \vdash_t \mathit{\Pi}_1[x := w] \cup \{x = (y.f)[x := w], (y.f)[x := w]\}$$

- Suppose $k = \texttt{NoCheck}$. Then $\mathit{\Pi}_1 \vdash y.f$, and so α contains a previous checked access to $\rho.f$ in the current span with no later releases. Hence

$$H; (\sigma' \uplus \sigma''); (\alpha \cdot a) \vdash_t \mathit{\Pi}_1[x := w] \cup \{x = (y.f)[x := w], (y.f)[x := w]\}$$

– $s = (y.f^k \texttt{ = } x)$: In this case,

$$\begin{array}{l} s' = \texttt{skip} \\ \sigma(x) = v \\ \sigma(y) = \rho \\ H' = H[\rho.f = v] \\ \sigma' = \sigma \\ a = t\!:\!\texttt{acc}(\rho.f^k) \\ k = \texttt{NoCheck} \Rightarrow \mathit{\Pi}_1 \vdash y.f \\ \mathit{\Pi}_2 = \mathit{\Pi}_1 \setminus f \cup \{x = y.f, y.f\} \end{array}$$

Let $\mathit{\Pi}_3 = \mathit{\Pi}_4 = \mathit{\Pi}_2$ and $\sigma'' = \emptyset$. Rule [Skip] shows that $\overline{D} \vdash s' : \mathit{\Pi}_3 \leadsto \mathit{\Pi}_4$. We must show that $H; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_2$. Since $H; \sigma; \alpha \vdash_t \mathit{\Pi}_1$, it must be that:

$$\begin{array}{ll} H; \sigma; \alpha \vdash_t p & \forall p \in \mathit{\Pi}_1 \\ H; \sigma; \alpha \vdash_t c & \forall c \in \mathit{\Pi}_1 \\ H; \sigma; \alpha \vdash_t \mathit{be} & \forall \mathit{be} \in \mathit{\Pi}_1 \end{array}$$

The following all hold:

- For all $p \in \mathit{\Pi}_1 \setminus f$, it must be that $H'; \sigma; \alpha \vdash_t p$ because $\mathit{\Pi}_1 \setminus f \subseteq \mathit{\Pi}_1$ and the rule [Path] is insensitive to the heap H.
- For all $c \in \mathit{\Pi}_1 \setminus f$, it must be that $H'; \sigma; \alpha \vdash_t c$ because no c in $\mathit{\Pi}_1 \setminus f$ will refer to an f field of any object in H'.
- For all $be \in \mathit{\Pi}_1 \setminus f$, it must be that $H'; \sigma; \alpha \vdash_t be$ because $\sigma(be) = \texttt{true}$.

Thus, $H'; \sigma; \alpha \vdash_t \mathit{\Pi}_1 \setminus f$ via rule [Context]. Lemma 3 then shows that $H'; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_1 \setminus f$.

Also, $\sigma(x) = v$ and $H'(\sigma(y.f)) = H'(\rho.f) = v$. Thus, we can show that $H'; \sigma; (\alpha \cdot a) \vdash_t (x = y.f)$.

We now consider the two cases for k:

- Suppose $k = \texttt{Check}$. Then $H'; \sigma; (\alpha \cdot a) \vdash_t y.f$ since $\sigma(y.f) = \rho.f$ and a is a checked access to $\rho.f$ by t. Thus,

$$H'; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_1 \setminus f \cup \{x = y.f, y.f\}$$

- Suppose $k = \texttt{NoCheck}$. Then $\mathit{\Pi}_1 \vdash y.f$, and so α contains a previous checked access to $\rho.f$ in the current span with no later releases. Hence

$$H'; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_1 \setminus f \cup \{x = y.f, y.f\}$$

Therefore, $H'; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}_2$.

– $s = (x \texttt{ = } y.m(\overline{z}))$: In this case, rule [Call] and the reduction rule for method calls require that:

$m(\overline{z'})\ \texttt{requires}\ P\ \texttt{ensures}\ Q\ \texttt{modifies}\ K\ \{\ s'';\ \texttt{return}\ r\ \} \in \overline{D}$
$\theta_s = [\overline{z' := z}, \texttt{this} := y, r := x]$
$\forall p \in \theta_s(P).\ \mathit{\Pi}_1 \vdash p$
$\mathit{\Pi}_2 = (\mathit{\Pi}_1 \setminus K)[x := w] \cup \theta_s(Q)$
θ_d maps $\mathrm{FV}(s'') \setminus \{r\}$ to fresh variables
θ_d maps $\overline{z'}, \texttt{this}, r$ to $\overline{z}, y, x$, respectively
$H' = H$
$\sigma' = \sigma$
$a = t\!:\!\epsilon$
$s' = \theta_d(s'')$

From rule [Method]:

m is unique in $\overline{D}$
s does not mutate $\texttt{this}$, $\overline{x}$
K contains all fields that may be modified while evaluating s''
K contains All if a lock may be released while evaluating s''
$\mathrm{FV}(P) \subseteq \{\texttt{this}, \overline{z'}\}$
$\mathrm{FV}(Q) \subseteq \{\texttt{this}, \overline{z'}, r\}$
$\overline{D} \vdash s'' : P \rightsquigarrow Q'$
$\forall q \in Q.\ Q' \vdash q$

Let

$$\begin{aligned}&\mathit{\Pi}_3 = \theta_d(P) \cup \mathit{\Pi}_1[x := w]\\&\mathit{\Pi}_4 = (\mathit{\Pi}_1 \setminus K)[x := w] \cup \theta_d(Q')\\&\sigma'' = [x := \sigma(x)]\end{aligned}$$

Then, $H; (\sigma \uplus \sigma''); \alpha \vdash_t \mathit{\Pi}_1[x := w]$. Also, θ_s and θ_d agree on $\mathrm{FV}(P)$, and $\mathit{\Pi}_1 \sqsubseteq \theta_s(P)$. So, $\mathit{\Pi}_1 \sqsubseteq \theta_d(P)$, and $H; (\sigma \uplus \sigma''); (\alpha \cdot a) \vdash_t \mathit{\Pi}_1[x := w] \cup \theta_d(P)$ follows by Lemma 3.
We next show that $\overline{D} \vdash \theta_d(s'') : \mathit{\Pi}_3 \leadsto \mathit{\Pi}_4$. Given $\overline{D} \vdash s'' : P \leadsto Q'$, it follows that:

$$\begin{aligned}&\overline{D} \vdash \theta_d(s'') : \theta_d(P) \leadsto \theta_d(Q') && \text{via Lemma 6}\\&\overline{D} \vdash \theta_d(s'') : \theta_d(P) \cup \mathit{\Pi}_1[x := w]\\&\qquad\qquad \leadsto \theta_d(Q') \cup (\mathit{\Pi}_1[x := w]) \setminus K && \text{via Lemma 7}\end{aligned}$$

Finally, we show that $\mathit{\Pi}_4 \sqsubseteq \mathit{\Pi}_2$. Treating Q' and Q as contexts containing only paths, we have

$$\begin{aligned}&Q' \sqsubseteq Q\\&\theta_d(Q') \sqsubseteq \theta_d(Q)\\&\theta_d(Q') \sqsubseteq \theta_s(Q) \qquad \text{since } \theta_d \text{ and } \theta_s \text{ agree on all variables in } \mathrm{FV}(Q) \cup \mathrm{FV}(Q')\\&(\mathit{\Pi}_1 \setminus K)[x := w] \cup \theta_d(Q') \sqsubseteq (\mathit{\Pi}_1 \setminus K)[x := w] \cup \theta_s(Q)\end{aligned}$$

□

We now state several technical lemmas used in the arguments above. First, any context well-formed under a run-time heap H, thread store σ, and trace α is also well-formed when α is extended with any non-span-ending operation.

Lemma 3. *If $H; \sigma; \alpha \vdash_t \mathit{\Pi}$ and a is a non-release action of thread t then $H; \sigma; (\alpha \cdot a) \vdash_t \mathit{\Pi}$.*

Proof. Proof is by case analysis for a and the definition of well-formed contexts. □

In addition, extending a well-formed context with a true boolean expression yields a well-formed context.

Lemma 4. *If $H; \sigma; \alpha \vdash_t \mathit{\Pi}$ and $\sigma(be) = \texttt{true}$ then $H; \sigma; \alpha \vdash_t \mathit{\Pi} \cup \{be\}$.*

Proof. This follows from the definition of well-formed contexts. □

The type system also enjoys the following monotonicity and substitution properties.

Lemma 5 (Monotonicity). *If $\overline{D} \vdash s : \mathit{\Pi}_1 \leadsto \mathit{\Pi}_2$ and $\mathit{\Pi}_3 \sqsubseteq \mathit{\Pi}_1$, then there exists $\mathit{\Pi}_4$ such that $\overline{D} \vdash s : \mathit{\Pi}_3 \leadsto \mathit{\Pi}_4$ and $\mathit{\Pi}_4 \sqsubseteq \mathit{\Pi}_2$.*

Proof. Proof is by induction on the derivation of $\overline{D} \vdash s : \mathit{\Pi}_1 \leadsto \mathit{\Pi}_2$. □

Lemma 6 (Substitution). *If* $\overline{D} \vdash s : \Pi_1 \leadsto \Pi_2$ *and* $\theta : Var \rightarrow Var$ *is a permutation on variables, then* $\overline{D} \vdash \theta(s) : \theta(\Pi_1) \leadsto \theta(\Pi_2)$

Proof. Proof is by induction on the derivation of $\overline{D} \vdash s : \Pi_1 \leadsto \Pi_2$. □

The following technical lemma is used in the proof that typing is preserved by the evaluation step for method invocation.

Lemma 7 (Extension). *Suppose*

$\overline{D} \vdash s : \Pi_1 \leadsto \Pi_2$
$\mathrm{FV}(\Pi_3)$ *is disjoint from the variables modified while evaluating s*
K *contains all fields that may be modified while evaluating s*
K *contains* $\textsc{All}$ *if a lock may be released while evaluating s*

Then $\overline{D} \vdash s : \Pi_1 \cup \Pi_3 \leadsto \Pi_2 \cup (\Pi_3 \setminus K)$.

Proof. Proof is by induction on the derivation of $\overline{D} \vdash s : \Pi_1 \leadsto \Pi_2$. □

Given Theorem 2, we can now prove that any verified program state Σ generates only well formed traces α. This shows that a race detector can safely ignore `NoCheck` accesses in any verified program, with no loss in precision.

Restatement of Theorem 3 (Soundness). *If* $\overline{D}; \epsilon \vdash \Sigma$ *and* $\Sigma \longrightarrow^{\alpha} \Sigma'$, *then* α *is well-formed.*

Proof. Proof is by induction on α using Lemma 8.

Lemma 8. *If* α *is well-formed and* $\overline{D}; \alpha \vdash \Sigma$ *and* $\Sigma \longrightarrow^{a} \Sigma'$, *then* $(\alpha \cdot a)$ *is well-formed.*

Proof. If a is not an unchecked access, then $\alpha \cdot a$ is trivially well-formed. If a is an unchecked access $t\!:\!\mathtt{acc}(\rho.f^{\mathtt{NoCheck}})$ then we must show that

$$\alpha = \alpha_1 \cdot t\!:\!\mathtt{acc}(\rho.f^{\mathtt{Check}}) \cdot \alpha_2$$

for some α_1 and α_2 such that α_2 contains no actions of the form $t : \mathtt{rel}(x)$, under the assumption that α is race-free. Assume a is a read action. (Writes are similar.) From the reduction rule,

$\Sigma = H \bullet T$
$T(t) = \langle \sigma, \mathcal{E}[x\ \mathtt{=}\ y.f^{\mathtt{NoCheck}}] \rangle$
$\sigma(y) = \rho$

Then,

$$\overline{D}; \alpha \vdash_t \langle \sigma, \mathcal{E}[x\ \mathtt{=}\ y.f^{\mathtt{NoCheck}}] \rangle$$

via rule [State], and rule [Thread] requires that

$\overline{D} \vdash \mathcal{E}[x\ \mathtt{=}\ y.f^{\mathtt{NoCheck}}] : \Pi \leadsto \Pi'$
$H; \sigma'; \alpha \vdash_t \Pi$

where σ' extends σ with a mapping for the Skolem variables in Π.

Lemma 9 (below) then shows that

$$\overline{D} \vdash x \texttt{ = } y.f^{\texttt{NoCheck}} : \Pi \leadsto \Pi''$$

Rule [Read] requires that $\Pi \vdash y.f$. Property 1 then requires that $H; \sigma'; \alpha \vdash_t \{y.f\}$, which means that $H; \sigma'; \alpha \vdash_t y.f$. Thus,

$$\alpha = \alpha_1 \cdot t\colon\texttt{acc}(\rho.f^{\texttt{Check}}) \cdot \alpha_2$$

where α_2 contains no actions of the form $t\colon\texttt{rel}(x)$, and $\alpha \cdot a$ is therefore well-formed. □

The following lemma states that any statement appearing in the evaluation context of a larger well-typed statement is also typeable from the same pre-context.

Lemma 9. *If $\overline{D} \vdash \mathcal{E}[s] : \Pi \leadsto \Pi'$ then there exists Π'' such that $\overline{D} \vdash s : \Pi \leadsto \Pi''$.*

Proof. By induction on the structure of $\mathcal{E}$. □