Conditional Must Not Aliasing for Static Race Detection

Mayur Naik   Alex Aiken

Computer Science Department
Stanford University
{mhn,aiken}@cs.stanford.edu

Abstract

Race detection algorithms for multi-threaded programs using the common lock-based synchronization idiom must correlate locks with the memory locations they guard. The heart of a proof of race freedom is showing that if two locks are distinct, then the memory locations they guard are also distinct. This is an example of a general property we call *conditional must not aliasing*: Under the assumption that two objects are not aliased, prove that two other objects are not aliased. This paper introduces and gives an algorithm for conditional must not alias analysis and discusses experimental results for sound race detection of Java programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — Reliability

General Terms Experimentation, Reliability, Verification

Keywords static race detection, Java, synchronization, concurrency, multi-threading

1. Introduction

A multi-threaded program contains a *race* if two threads can access the same memory location without ordering constraints enforced between them and at least one of those accesses is a write. Races often imply serious violations of program invariants and are notoriously difficult to find in debugging and testing. Proving *race freedom*—the absence of races—is thus valuable in improving the reliability of multi-threaded programs.

Most approaches to proving race freedom involve checking the *lock-based* synchronization idiom [3, 4, 15, 16, 22, 36, 39]. Locking requires that any pair of potentially simultaneous accesses to a location m from different threads be *guarded* by a lock l, meaning that each thread must hold lock l while accessing m. Because at most one thread can hold lock l at any instant, there are no races on m if the locking discipline is used correctly.

A challenge in proving race freedom in the presence of locks lies in the apparent need for a form of must alias analysis. Consider the following pseudo-code example:

```
Thread 1: sync(l1) { ... write location m1 ... }

Thread 2: sync(l2) { ... write location m2 ... }
```

Here `sync` is Java's lexically-scoped locking construct. The lock argument to `sync` is acquired before entering the block and released on exiting the block. Consider the two memory accesses in this example, and suppose that it is possible that `m1` $=$ `m2` at run-time, i.e., `m1` and `m2` *may alias*. To show that the accesses cannot race, it suffices to show that `l1` and `l2` always refer to the same lock at run-time, i.e., `l1` and `l2` *must alias*.

In previous work [33], we presented a static race detection technique for Java employing a series of static analyses to successively prune an initial over-estimate of the set of races to a relatively small set of potential races. The analysis of locks used is an approximation based on a may alias analysis to check whether a pair of locks held during a pair of accesses might be the same. This approximation, while effective at bug-finding, does not perform the needed must alias analysis and cannot prove race freedom. It merely partitions races into *likely* and *unlikely* races. While the likely races are in practice almost always real races, the set of unlikely races has a high false positive rate and it is difficult to know how many real races it contains (if any) without considerable manual labor.

In this paper, we present a new approach to sound race detection in the presence of locks. The key idea is that instead of attacking the problem directly using a must alias analysis, we reformulate the question that must be solved in terms of a dual *must not alias* analysis problem. Consider once more the example above. Instead of starting with the locations accessed and reasoning about the locks, we start with the locks and try to reason about the locations. If we assume that locks `l1` and `l2` cannot be the same (we assume the locks must not alias) then it suffices to show that the locations `m1` and `m2` are not the same (we prove the locations must not alias). Intuitively, if whenever two locks are different their guarded locations must be different, then there are no races. Note that in the case where the two locks must alias there is nothing to prove—accesses to their guarded locations cannot race in any case.

This approach to proving race freedom is an example of a *conditional must not alias* query:

DEFINITION 1.1. Let e_1 and e_2 be abstract memory locations (e.g., program expressions with associated context sensitive information). A must not alias fact is a pair (e_1, e_2) asserting e_1 and e_2 cannot refer to the same run-time memory location. Let P be a program and A and B be sets of must not alias facts. A *conditional must not alias* sentence $A \vdash_P B$ means that in program P, under the assumption that the must not alias facts in A hold, the must not alias facts in B must hold.

We use the example in Figure 1 to illustrate the idea of must not alias analysis in the context of race detection. This example has three parts shown in three columns. The code in the first column allocates an array object h_1 and executes a loop, each iteration of which puts a fresh object h_2, whose field `f` points to another fresh object h_3, into the array. Next, the code in the second column executes a loop, each iteration of which spawns a thread that accesses

POPL'07 January 17–19, 2007, Nice, France.

field g of an object h_3 through field f of a non-deterministically chosen array element object h_2. Left unspecified is the lock L that is acquired; several choices for L are given in the third column.

Consider the case where L = a: the lock is acquired on the entire array. This situation represents a coarse-grain locking style that uses global, uniquely named locks; in particular, each such lock is created at an allocation site that executes exactly once. Some previous sound lock checking systems rely on such single execution allocation sites for locks [5]. From the point of view of conditional must not aliasing this case is uncomplicated. Consider two syncs in different iterations of the second loop. Since the assumption that they acquire different locks is false (they always acquire the same lock), we can conclude that whenever the locks are different the guarded accesses are distinct and the program is race free.

Now consider the case where L = x.f. This case represents the extreme of fine-grain locking, and again reasoning using conditional must not aliasing to prove race freedom is straightforward. If two syncs in different iterations of the second loop acquire different locks, then the locations of their g fields must be different and the code is race free.

The subtlest case is medium-grain locking represented by the case L = x. Each iteration of the loop holds a lock on object x, but the potential race is on field g of a different object x.f. The key to showing this example is race free lies in observing that x.f is only reachable through x and therefore locking x is sufficient to guard against races on fields of x.f. Thus, if in two different iterations of the loop the x objects are different, then the x.f objects (and hence the locations of their g fields) must also be different and the code is race free.

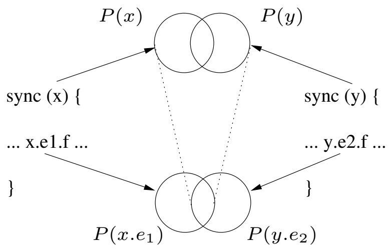
All three of these locking styles (coarse-, medium-, and fine-grain) occur frequently in real programs. Note that a common theme in the informal arguments given above is the ability to reason about different locks acquired at the same syntactic point, as alluded to in phrases such as "if two syncs in different iterations . . . ". The medium-grain locking case has the additional difficulty of reasoning about correlated objects, such as the fact that in the first loop, field f of each new h_2 object points to a unique h_3 object allocated in the same loop iteration. We develop a type and effect system that tracks the needed correlations between objects. Section 2 introduces a small language we use for the formal development and Section 3 introduces the type and effect system and gives a proof of its soundness.

To concisely represent must not aliasing facts, we use a separate object reachability analysis we call *disjoint reachability analysis* (Section 4). Let $\mathbb{H}$ be the set of all allocation sites in the program; we define a function $\mathit{DR} \in 2^{\mathbb{H}} \to 2^{\mathbb{H}}$ such that if $h \in \mathit{DR}(H)$, then any object o allocated at site h is reachable by following one or more field dereferences from at most one of any two distinct objects o_1 and o_2 allocated at sites $h_1, h_2 \in H$. Note that we require $o_1 \neq o_2$, but we allow $h_1 = h_2$. In the example in Figure 1, we have $h_3 \in \mathit{DR}(\{h_2\})$. Figure 2 gives a pictorial view of how disjoint reachability analysis is used to prove race freedom. In Figure 2, $P(e)$ is the *points-to set* of e, i.e., the set of allocation sites at which e may be allocated. Let x and y be locks and $x.e_1.f$ and $y.e_2.f$ be two accesses to instance field f. Now, $P(x.e_1) \cap P(y.e_2)$ is the set of may aliases of $x.e_1$ and $y.e_2$—the set of objects on which there is a potential race. If this set is contained in $\mathit{DR}(P(x) \cup P(y))$, however, then whenever x and y are distinct objects it is guaranteed that $x.e_1$ and $y.e_2$ are distinct objects and no races are possible.

We have implemented the type and effect system and disjoint reachability analysis in Chord (Section 5), a static race detector for Java and performed a number of experiments on complete Java applications (see Section 6). The main empirical result is that our sound race detector eliminates almost all of the false positives in our previous work; the false positives remaining, at least in these benchmarks, are the result of engineering shortcomings that are straightforward to remove with additional effort. In particular, the large category of unlikely races is almost completely eliminated. Significantly, the number of races found in these benchmarks increases from 122 using our old unsound approach of considering only likely races to 202 using our new sound algorithm. Thus, a number of the former unlikely races turn out to be real races that we did not notice with our previous technique.

```
a = new h1[N];              while (*) {        Choices for L:
for (i = 0; i < N; i++) {     x = a[*];              a
  a[i] = new h2;              fork {                 x
  a[i].f = new h3;              sync (L) {           x.f
}                                 x.f.g = *;
                                } } }
```

Figure 1. Example program.



$$(P(x.e_1) \cap P(y.e_2)) \subseteq DR(P(x) \cup P(y)) \Rightarrow \text{(f is race-free)}$$

Figure 2. Proving race freedom via conditional must not aliasing.

In summary, the main contributions of this paper are:

- We introduce conditional must not aliasing, a property that is useful in formulating static race detection and may be of independent interest. Conditional must not aliasing is different from standard may aliasing precisely in that it is conditional; instead of computing must not aliasing facts that always hold, we compute must not aliasing facts that only hold assuming other must not aliasing facts, allowing a more refined treatment of the relationship between locks and the locations they guard.
- We introduce disjoint reachability analysis, a program analysis useful for computing conditional must not alias properties. Disjoint reachability analysis is also a cheaper, and likely more scalable (but less precise) alternative to some recent decision procedures for verification of pointer-based data structures. Further discussion is included with related work (Section 7).
- We have implemented conditional must not aliasing using a disjoint reachability analysis in Chord, a static race checker [33]. On the benchmarks we have studied, this system has few false positives and, because it is designed to be sound, should have no false negatives (modulo some standard unsound assumptions discussed in Section 5).

2. Language

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

2.1 Syntax

The abstract syntax of the language is given in Figure 3. A program has a fixed set of variables $\mathbb{V}$ with global scope and a single object type with instance fields $\mathbb{F}$. We label each object allocation site in the program with a unique $h \in \mathbb{H}$ and we label each loop with a unique integer $w \in \mathbb{W}$. There are no threads; conditional must not aliasing is not a multi-threading property and the presentation is simplest in a single-threaded language.

$$\begin{array}{rccl}
\textit{(variable)} & x, y & \in & \mathbb{V} \\
\textit{(instance field)} & f & \in & \mathbb{F} \\
\textit{(object allocation site)} & h & \in & \mathbb{H} \\
\textit{(loop identifier)} & w & \in & \mathbb{W}
\end{array}$$

$$\begin{array}{rcl}
s & ::= & x = \mathsf{null} \mid x = \mathsf{new}\, h \mid x = y \mid x = y.f \mid x.f = y \mid \\
 & & s_1 \,;\, s_2 \mid \mathsf{if}\,(*)\,\mathsf{then}\, s_1\, \mathsf{else}\, s_2 \mid \mathsf{while}^w\,(*)\,\mathsf{do}\, s
\end{array}$$

Figure 3. Abstract syntax.

$$\begin{array}{rcccl}
\textit{(loop iteration)} & \mathbb{N} & \ni & i & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
\textit{(loop vector)} & & & \pi & \in & \mathbb{W} \to \mathbb{N} \\
\textit{(non-null object)} & \mathbb{O} & \ni & \bar{o} & ::= & \langle \pi, h \rangle \\
\textit{(object)} & \mathbb{O}_\perp & \ni & o & ::= & \bar{o} \mid \perp \\
\textit{(environment)} & & & \rho & \in & \mathbb{V} \to \mathbb{O}_\perp \\
\textit{(heap)} & & & \sigma & \in & (\mathbb{O} \times \mathbb{F}) \to \mathbb{O}_\perp \\
\textit{(heap effect)} & & & C & ::= & \emptyset \mid C \cup \{\bar{o}_1 \rhd \bar{o}_2\}
\end{array}$$

$$\begin{array}{rcl}
\langle \pi, h \rangle.\boldsymbol{\pi} & \triangleq & \pi \\
\langle \pi, h \rangle.\boldsymbol{h} & \triangleq & h
\end{array}$$

Figure 4. Semantic domains.

2.2 Semantics

We next develop an operational semantics for the WHILE language in Figure 3. Figure 4 defines the semantic domains. Recall that one goal is to track reachability properties of objects (e.g., the example in Figure 1). Reasoning about object reachability requires reasoning about how data structures are built, which means reasoning about the times when objects are allocated and linked to one another. A *loop vector* π, which is a tuple of non-negative integers, tracks how many times each loop in a program has executed; each element of π is a counter, and the iteration count of a loop $\mathsf{while}^w\,(*)\,\mathsf{do}\, s$ in the program is $\pi(w)$ (treating the tuple π as a map from indices to elements of π). Objects are uniquely identified as pairs $\langle \pi, h \rangle$ of a loop vector π recording the time (in loop execution counts) when the object was allocated and its allocation site h. Abstractions of the loop vector (Section 3) will allow us to estimate the relative time in terms of the number of loop iterations when two distinct objects are allocated.

Environments and heaps are standard. An environment maps variable names to objects in the heap, and a heap records for each object what object (or null) is in each field. A *heap effect* $\bar{o}_1 \rhd \bar{o}_2$ records that at some point in the execution, object $\bar{o}_2$ was reachable in one step via a field dereference from object $\bar{o}_1$.

Figure 5 presents a big-step operational semantics for our language. Judgments have the form

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

Each step of execution begins with the statement s to be executed, the set W of all loops lexically enclosing s, and the current loop vector π, environment ρ, and heap σ. Note that W records which loops are currently executing while π records the execution count of all loops in the program and not just of loops currently active. Since loops may execute as part of a step of execution the semantics must record a new loop vector as well as an updated environment and heap. Thus, a step of execution terminates with a final loop vector π', environment ρ', and heap σ', plus the heap effects C.

$$\begin{array}{rcccl}
\textit{(loop iteration abstr.)} & \mathbb{N}_\top & \ni & \hat{i} & ::= & 0 \mid 1 \mid \top \\
\textit{(loop vector abstr.)} & & & \Pi & \in & \mathbb{W} \to \mathbb{N}_\top \\
\textit{(obj. alloc. site abstr.)} & \mathbb{H}_\top & \ni & \hat{h} & ::= & h \mid \top \\
\textit{(non-null type)} & \mathbb{T} & \ni & \bar{\tau} & ::= & \langle \Pi, \hat{h} \rangle \\
\textit{(type)} & \mathbb{T}_\perp & \ni & \tau & ::= & \bar{\tau} \mid \perp \\
\textit{(type environment)} & & & \Gamma & \in & \mathbb{V} \to \mathbb{T}_\perp \\
\textit{(heap effect abstr.)} & & & K & ::= & \emptyset \mid K \cup \{\bar{\tau}_1 \trianglerighteq \bar{\tau}_2\}
\end{array}$$

$$\begin{array}{rcl}
\langle \Pi, \hat{h} \rangle.\boldsymbol{\Pi} & \triangleq & \Pi \\
\langle \Pi, \hat{h} \rangle.\boldsymbol{\hat{h}} & \triangleq & \hat{h}
\end{array}$$

Figure 6. Types.

We explain the most interesting rules in Figure 5. Rule (2), which creates a new object, does not simply use the current loop vector as the time stamp recorded in the object. Instead, counters for loops not in W (i.e., those not currently executing) are set to 0, giving a way to determine later whether or not a particular loop was executing when the object was allocated.[1] While this property is not exploited in the instrumented operational semantics, it is used in the abstractions discussed in Section 3. Assigning to a field of an object (Rule (5)) generates a heap effect recording the reachability between the two objects involved in the assignment. Finally, consider Rules (9) and (10), which give the semantics of `while` statements. Loops execute a non-deterministic number of times, which saves us the trouble of defining how loop termination conditions (as well as the predicates of `if` statements, see Rules (6) and (7)) are evaluated. Also note that when a loop executes an additional time (Rule (10)) the appropriate loop counter is incremented.

We conclude this section with a small example, which is a simplified version of the program in Figure 1.

EXAMPLE 2.1. while^1 (*) do { x = new h_1; y = new h_2; x.f = y }
If the loop executes one time, the root of the derivation tree is:

$$\begin{array}{l}
\mathsf{while}^1 \ldots, \emptyset, \langle 0 \rangle, [x \mapsto \perp, y \mapsto \perp], [] \vdash \\
\quad \langle 1 \rangle, [x \mapsto \bar{o}_1, y \mapsto \bar{o}_2], [\bar{o}_1.\mathtt{f} \mapsto \bar{o}_2, \bar{o}_2.\mathtt{f} \mapsto \perp], \{\bar{o}_1 \rhd \bar{o}_2\}
\end{array}$$

where $\bar{o}_1 = \langle \langle 1 \rangle, h_1 \rangle$ and $\bar{o}_2 = \langle \langle 1 \rangle, h_2 \rangle$ and [] is the empty heap.

3. Type and Effect System

The syntax of types and effects is shown in Figure 6. Types and effects are parallel with the definitions in Figure 4, but the semantics are significantly different. Before proceeding with the formal development, we provide an informal explanation.

Objects have types $\langle \Pi, \hat{h} \rangle$ recording information about when and where they were allocated. The main purpose of the type system is to compute abstract heap effects such as $\langle \Pi_1, \hat{h_1} \rangle \trianglerighteq \langle \Pi_2, \hat{h_2} \rangle$. As in the operational semantics, the effect implies an object of type $\langle \Pi_2, \hat{h_2} \rangle$ is reachable from an object of type $\langle \Pi_1, \hat{h_1} \rangle$ in a single step via a field dereference. In a type, loop iterations are abstracted as 0, 1, or $\top$. If $\Pi_1(w) = 0$, then loop w was not executing when the object with that type was allocated (similarly for $\Pi_2(w)$). If $\Pi_1(w) = \top$ then nothing is known about the iteration of loop w in which the object was allocated (and similarly for $\Pi_2(w)$). In either case nothing is known about the relative time at which objects of the two types were allocated. However, if $\Pi_1(w) = \Pi_2(w) = 1$, then the type system guarantees the two objects were allocated in the same iteration of loop w. This property allows us to show conditional must not aliasing: intuitively, if $\langle \Pi_1, \hat{h_1} \rangle$ reaches $\langle \Pi_2, \hat{h_2} \rangle$ and they were allocated in the same iteration, then objects of type

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

$$x = \mathsf{null}, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho[x \mapsto \bot], \sigma, \emptyset \tag{1}$$

$$x = \mathsf{new}\ h, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho[x \mapsto \bar{o}], \sigma[\bar{o}.f_1 \mapsto \bot, ..., \bar{o}.f_n \mapsto \bot], \emptyset \qquad [\bar{o} = \langle \lambda w.(\text{if } w \in W \text{ then } \pi(w) \text{ else } 0), h\rangle] \tag{2}$$

$$x = y, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho[x \mapsto \rho(y)], \sigma, \emptyset \tag{3}$$

$$x = y.f, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho[x \mapsto \sigma(\bar{o}.f)], \sigma, \emptyset \qquad \text{if } \rho(y) = \bar{o} \tag{4}$$

$$x.f = y, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho, \sigma[\bar{o}_1.f \mapsto o_2], C \qquad \text{if } \rho(x) = \bar{o}_1 \qquad \left[\rho(y) = o_2 \text{ and } C = \begin{cases} \{\bar{o}_1 \rhd \bar{o}_2\} & \text{if } o_2 = \bar{o}_2 \\ \emptyset & \text{if } o_2 = \bot \end{cases}\right] \tag{5}$$

$$\frac{s_1, W, \pi, \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C}{\mathsf{if}\ (*)\ s_1\ \mathsf{else}\ s_2, W, \pi, \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C} \quad (6) \qquad\qquad \frac{s_2, W, \pi, \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C}{\mathsf{if}\ (*)\ s_1\ \mathsf{else}\ s_2, W, \pi, \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C} \quad (7)$$

$$\frac{s_1, W, \pi, \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C_1 \qquad s_2, W, \pi', \rho', \sigma' \ \Downarrow\ \pi'', \rho'', \sigma'', C_2}{s_1;\ s_2, W, \pi, \rho, \sigma \ \Downarrow\ \pi'', \rho'', \sigma'', C_1 \cup C_2} \tag{8}$$

$$\mathsf{while}^w\ (*)\ \mathsf{do}\ s, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho, \sigma, \emptyset \tag{9}$$

$$\frac{s, W \cup \{w\}, \pi[w \mapsto \pi(w)+1], \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C_1 \qquad \mathsf{while}^w\ (*)\ \mathsf{do}\ s, W, \pi', \rho', \sigma' \ \Downarrow\ \pi'', \rho'', \sigma'', C_2}{\mathsf{while}^w\ (*)\ \mathsf{do}\ s, W, \pi, \rho, \sigma \ \Downarrow\ \pi'', \rho'', \sigma'', C_1 \cup C_2} \tag{10}$$

Figure 5. Instrumented operational semantics.

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

(a) Object abstraction.

$$C \preceq K \quad \Leftrightarrow \quad \forall(\bar{o}_1 \rhd \bar{o}_2) \in C :\ \exists(\bar{\tau}_1 \trianglerighteq \bar{\tau}_2) \in K :\ (\bar{o}_1, \bar{o}_2) \propto (\bar{\tau}_1, \bar{\tau}_2)$$

$$(\bar{o}_1, \bar{o}_2) \propto (\bar{\tau}_1, \bar{\tau}_2) \quad \Leftrightarrow \quad \begin{cases} & (1)\ \ \bar{o}_1 \preceq \bar{\tau}_1 \\ \wedge & (2)\ \ \bar{o}_2 \preceq \bar{\tau}_2 \\ \wedge & (3)\ \ \forall w \in \mathbb{W} :\ ((\bar{\tau}_1.\mathbf{\Pi}(w) = 1 \wedge \bar{\tau}_2.\mathbf{\Pi}(w) = 1) \Rightarrow \bar{o}_1.\boldsymbol{\pi}(w) = \bar{o}_2.\boldsymbol{\pi}(w)) \end{cases}$$

(b) Heap effect abstraction.

$$W \vdash (\pi, \rho) \preceq (\Pi, \Gamma) \quad \Leftrightarrow \quad \begin{cases} & (1)\ \forall w \in W :\ \pi(w) \preceq \Pi(w) \\ \wedge & (2)\ \forall x \in \mathbb{V} :\ \rho(x) \preceq \Gamma(x) \\ \wedge & (3)\ \forall w \in \mathbb{W} :\ \exists k \in \mathbb{N} : \begin{cases} & (a)\ \Pi(w) = 1 \Rightarrow \pi(w) = k \\ \wedge & (b)\ \forall x \in \mathbb{V} :\ ((\rho(x) = \bar{o} \wedge \Gamma(x) = \bar{\tau} \wedge \bar{\tau}.\mathbf{\Pi}(w) = 1) \Rightarrow \bar{o}.\boldsymbol{\pi}(w) = k) \end{cases} \end{cases}$$

(c) Environment abstraction.

Figure 7. Abstraction.

$\langle \Pi_1, \hat{h_1} \rangle$ allocated in different iterations must reach different objects of type $\langle \Pi_2, \hat{h_2} \rangle$.

This discussion is made precise in Figure 7, which defines an abstraction relation $\preceq$ stating when types and abstract heap effects abstract objects and concrete heap effects, respectively. The third clause of sub-figure (b) requires that the iteration counts in position w of the concrete loop vectors of two objects match if the values in position w of the abstract loop vectors of their types are 1. Likewise, the third clause of sub-figure (c) requires that iteration counts in position w of the concrete loop vectors of all objects in environment ρ match if the values in position w of the abstract loop vectors of their types in environment Γ are 1. Thus, in both abstract heap effects and type environments, any two types with a 1 in position w of their abstract loop vectors always abstract objects allocated in the same concrete, but unknown, iteration of loop w.

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

Definition 3.1. **(Join of Environments)**

$$\begin{array}{rcl}
(\Gamma_1 \sqcup \Gamma_2)(x) & = & \Gamma_1(x) \sqcup \Gamma_2(x) \\
\tau_1 \sqcup \tau_2 & = & \begin{cases} \tau_1 & \text{if } \tau_2 = \bot \\ \tau_2 & \text{if } \tau_1 = \bot \\ \bar{\tau}_1 \sqcup \bar{\tau}_2 & \text{if } \tau_1 = \bar{\tau}_1 \wedge \tau_2 = \bar{\tau}_2 \end{cases} \\
\bar{\tau}_1 \sqcup \bar{\tau}_2 & = & \langle \bar{\tau}_1.\mathbf{\Pi} \sqcup \bar{\tau}_2.\mathbf{\Pi}, \bar{\tau}_1.\hat{\boldsymbol{h}} \sqcup \bar{\tau}_2.\hat{\boldsymbol{h}} \rangle \\
(\Pi_1 \sqcup \Pi_2)(w) & = & \Pi_1(w) \sqcup \Pi_2(w) \\
\hat{i}_1 \sqcup \hat{i}_2 & = & \begin{cases} \hat{i}_1 & \text{if } \hat{i}_1 = \hat{i}_2 \\ \top & \text{otherwise} \end{cases} \\
\hat{h}_1 \sqcup \hat{h}_2 & = & \begin{cases} \hat{h}_1 & \text{if } \hat{h}_1 = \hat{h}_2 \\ \top & \text{otherwise} \end{cases}
\end{array}$$

DEFINITION 3.2. **(Loopback Environment)**

$$\begin{array}{rcl} \Gamma^{w+}(x) & = & \Gamma(x)^{w+} \\ \bar{\tau}^{w+} & = & \langle \bar{\tau}.\mathbf{\Pi}^{w+}, \bar{\tau}.\boldsymbol{\hat{h}} \rangle \\ \bot^{w+} & = & \bot \\ \Pi^{w+}(w') & = & \left\{ \begin{array}{ll} \top & \text{if } w' = w \ \wedge \ \Pi(w) = 1 \\ \Pi(w') & \text{otherwise} \end{array} \right. \end{array}$$

The upper bound of two types implicitly defines a type lattice, which is ordered pointwise on loop vectors and allocation sites. Integers and allocation sites are all less than $\top$ and incomparable to each other. The maximal type is then a loop vector of all $\top$ elements and a $\top$ allocation site; the minimal element is the type $\bot$, and since any program has a finite number of loops and allocation sites, the type lattice is also finite.

The type rules are given in Figure 8. These rules are parallel with the operational semantics in Figure 5 and for brevity we point out only a few interesting features. Rule (12) puts 0's in the loop vector positions of newly allocated objects for any loops that are not executing, just as in Rule (2) of Figure 5. The only use of W is to distinguish active from inactive loops; loop vector positions of active loops take their value from the current loop vector Π. Rule (14) gives no information about heap reads, which is sound, but overly conservative in practice. We discuss improvements in Section 5, which we omit from the formal development for simplicity. The most interesting rule, Rule (18), has three important aspects. First, the condition $\Pi(w) \neq 0$ reflects that loop vectors for objects allocated inside loop w should not be 0 at position w (and Rule (12) already guarantees objects allocated outside loop w have a 0 at position w). Second, the fact that the environment Γ is the same before and after the loop reflects that any conclusion must be valid for any number of executions of the loop—that is, the entire loop may be executed multiple times (e.g., if it is nested inside of another loop) and the environment Γ must be an invariant for all of those executions. For example, a proof $W, \Pi, \Gamma \ \vdash \ \mathsf{while}^{w} \ \ldots : \Gamma, K$ where $\Gamma = [x \mapsto \langle\langle \ldots, 1, \ldots \rangle, h_1\rangle, y \mapsto \langle\langle \ldots, 1, \ldots \rangle, h_2\rangle]$ implies that if the loop ever starts execution in an environment where x and y were allocated in the same iteration of some earlier execution of the loop (e.g., because this loop is nested inside another loop), then the loop terminates with x and y assigned objects from the same loop iteration. Note that the final concrete loop iteration associated with x and y may be different than the initial one; the value 1 in both types only requires that the concrete loop iterations of x and y be equal before and after the loop, but the loop may assign new objects to x and y from the same iteration and maintain this property. Third and finally, any objects in the environment at the start of a loop iteration must be carried over from previous iterations. Thus, the body s of loop w is checked in the environment Γ^{w+}, which ensures that objects in the environment at the start of a new iteration are not 1 in the w^{th} component of their loop vector; $\Pi(w)$, however, can be 1, which allows any objects s allocates to be recognized as allocated together in the same iteration.

As an aside, for a single loop the only correlation this type system can recognize is when objects are allocated and linked in the same iteration of the loop (Rule (15)). By adding more abstract loop vector values (i.e., 2,3,4, ...) and adjusting definitions (e.g., Definition 3.2) the system can be extended to recognize when a value is allocated in one iteration and linked to an object allocated in the next iteration, or two iterations later, and so on. However, so far we have not found this extra power necessary, at least for race detection, and so we have presented and implemented the simpler system. Much more important is correctly handling multiple nested (and non-nested) loops and this is the focus of our system.

We are now ready to state the type preservation lemma. Appendix A gives a proof of the key cases.

LEMMA 3.3. **(Type Preservation)** If $s, W, \pi, \rho, \sigma \ \Downarrow \ \pi', \rho', \sigma', C$ and $W, \Pi, \Gamma \ \ \vdash \ \ s : \Gamma', K$ and $W \ \vdash \ (\pi, \rho) \ \preceq \ (\Pi, \Gamma)$ then $W \vdash (\pi', \rho') \preceq (\Pi, \Gamma')$ and $C \preceq K$.

Recall that the purpose of the type system is to compute a set of heap effect abstractions, which we use in disjoint reachability analysis (Section 4). We use type preservation to prove the soundness of heap effect abstraction.

COROLLARY 3.4. **(Soundness of Heap Effect Abstraction)** If $s, \emptyset, \lambda w.0, \lambda x.\bot, [] \ \Downarrow \ \pi, \rho, \sigma, C$ and $\emptyset, \Pi, \Gamma \ \vdash \ s : \Gamma', K$ then $C \preceq K$.

Proof. From Figure 7(c), we have $\emptyset \vdash (\lambda w.0, \lambda x.\bot) \preceq (\Pi, \Gamma)$. It follows from Lemma 3.3 that $C \preceq K$. □

Returning to Example 2.1 at the end of Section 2, the type system can prove:

$\emptyset, \langle 1 \rangle, \Gamma \ \vdash \ \mathsf{while}^1 \ldots \ : \ \Gamma, \{\bar{\tau}_1 \trianglerighteq \bar{\tau}_2\}$ where
$\Gamma = [x \mapsto \bar{\tau}_1, y \mapsto \bar{\tau}_2]$, $\bar{\tau}_1 = \langle\langle 1 \rangle, h_1\rangle$, and $\bar{\tau}_2 = \langle\langle 1 \rangle, h_2\rangle$

EXAMPLE 3.5. Consider the following nested loop, with two possible statements (A) and (B) for the body of the inner loop:

```
while$^1$ (*) do
    x = new $h_1$;
    while$^2$ (*) do
        y = new $h_2$;
        (A) x.f = y  OR  (B) y.f = x
```

Statement (A) abstracts a typical programming pattern for containers: the outer object x controls access to objects y allocated in an inner loop (in realistic examples all y's would be retained in e.g., a list). With statement (A), the type system can prove:

$\emptyset, \langle 1, 1 \rangle, \Gamma \vdash \mathsf{while}^1 \ldots : \Gamma \ \{\bar{\tau}_1 \trianglerighteq \bar{\tau}_2\}$ where
$\Gamma = [x \mapsto \bar{\tau}_1, y \mapsto \bar{\tau}_2]$, $\bar{\tau}_1 = \langle\langle 1, 0 \rangle, h_1\rangle$, and $\bar{\tau}_2 = \langle\langle 1, 1 \rangle, h_2\rangle$

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

$\emptyset, \langle 1, 1 \rangle, \Gamma \vdash \mathsf{while}^1 \ldots : \Gamma \ \{\bar{\tau}_2 \trianglerighteq \bar{\tau}_1\}$ where
$\Gamma = [x \mapsto \bar{\tau}_1, y \mapsto \bar{\tau}_2]$, $\bar{\tau}_1 = \langle\langle 1, 0 \rangle, h_1\rangle$, and $\bar{\tau}_2 = \langle\langle 1, 1 \rangle, h_2\rangle$

4. Disjoint Reachability Analysis

In this section, we present *disjoint reachability analysis*, an object reachability analysis used to compute conditional must not aliasing facts. We first formalize the notion of object reachability embodied in the *disjoint reachability property*. We then present disjoint reachability analysis which uses the heap effect abstraction K of a well-typed program to approximate the disjoint reachability property. Finally, we prove the disjoint reachability analysis sound with respect to the disjoint reachability property.

Consider the concrete heap effect C of a program execution; C contains an effect $(\bar{o}_1 \triangleright \bar{o}_2)$ if and only if some instance field f of object $\bar{o}_1$ was assigned object $\bar{o}_2$ during execution (recall Section 2.2). The (non-reflexive) transitive closure of C is $C^+ = \bigcup_{n \geq 1} C^n$, where C^n is:

DEFINITION 4.1. **(Closure of C)**

1. $C^1 = C$
2. If $(\bar{o}_1 \triangleright \bar{o}_2) \in C^n$ and $(\bar{o}_2 \triangleright \bar{o}_3) \in C$ then $(\bar{o}_1 \triangleright \bar{o}_3) \in C^{n+1}$.

If $(\bar{o}_1 \triangleright \bar{o}_2) \ \in \ C^n$, then $\bar{o}_2$ may be reachable from $\bar{o}_1$ by n field dereferences. The disjoint reachability property is given in the first equation in Figure 9. It says that $h \in \mathit{DR}_C(H)$ if and

$$W, \Pi, \Gamma \ \vdash \ x = \mathsf{null} : \Gamma[x \mapsto \bot], \emptyset \tag{11}$$

$$W, \Pi, \Gamma \ \vdash \ x = \mathsf{new}\, h : \Gamma[x \mapsto \langle \Pi', h \rangle], \emptyset \qquad \left[\Pi' = \lambda w.(\text{if } w \in W \text{ then } \Pi(w) \text{ else } 0)\right] \tag{12}$$

$$W, \Pi, \Gamma \ \vdash \ x = y : \Gamma[x \mapsto \Gamma(y)], \emptyset \tag{13}$$

$$W, \Pi, \Gamma \ \vdash \ x = y.f : \Gamma[x \mapsto \langle \lambda w.\top, \top \rangle], \emptyset \tag{14}$$

$$W, \Pi, \Gamma \ \vdash \ x.f = y : \Gamma, K \qquad \left[K = \begin{cases} \{\bar{\tau}_1 \trianglerighteq \bar{\tau}_2\} & \text{if } \Gamma(x) = \bar{\tau}_1 \text{ and } \Gamma(y) = \bar{\tau}_2 \\ \emptyset & \text{otherwise} \end{cases} \right] \tag{15}$$

$$\frac{W, \Pi, \Gamma \ \vdash \ s_1 : \Gamma', K_1 \qquad W, \Pi, \Gamma' \ \vdash \ s_2 : \Gamma'', K_2}{W, \Pi, \Gamma \ \vdash \ s_1; \ s_2 : \Gamma'', K_1 \cup K_2} \tag{16}$$

$$\frac{W, \Pi, \Gamma \ \vdash \ s_1 : \Gamma_1, K_1 \qquad W, \Pi, \Gamma \ \vdash \ s_2 : \Gamma_2, K_2}{W, \Pi, \Gamma \ \vdash \ \mathsf{if}\ (*)\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 : \Gamma_1 \sqcup \Gamma_2, K_1 \cup K_2} \tag{17}$$

$$\frac{W \cup \{w\}, \Pi, \Gamma^{w+} \ \vdash \ s : \Gamma, K}{W, \Pi, \Gamma \ \vdash \ \mathsf{while}^w\ (*)\ \mathsf{do}\ s : \Gamma, K} \qquad [\Pi(w) \neq 0] \tag{18}$$

Figure 8. Type rules.

$$h \in DR_C(H) \quad \Leftrightarrow \quad \left(\begin{array}{r} \bar{o}_1.\boldsymbol{h} \in H \ \wedge \ (\bar{o}_1 \triangleright \bar{o}) \in C^+ \quad \wedge \\ \bar{o}_2.\boldsymbol{h} \in H \ \wedge \ (\bar{o}_2 \triangleright \bar{o}) \in C^+ \quad \wedge \\ \bar{o}.\boldsymbol{h} = h \end{array} \ \Rightarrow \ \bar{o}_1 = \bar{o}_2 \right)$$

$$h \in DR_K(H) \quad \Leftrightarrow \quad \left(\begin{array}{r} (\bar{\tau}_1 \trianglerighteq \bar{\tau}_3) \in K^+ \quad \wedge \\ (\bar{\tau}_2 \trianglerighteq \bar{\tau}_4) \in K^+ \quad \wedge \\ \bar{\tau}_3 \sim \bar{\tau}_4 \quad \wedge \\ \bar{\tau}_3.\hat{\boldsymbol{h}}, \bar{\tau}_4.\hat{\boldsymbol{h}} \in \{h, \top\} \quad \end{array} \Rightarrow \begin{array}{l} \bar{\tau}_1.\hat{\boldsymbol{h}} \neq \top \wedge \\ \bar{\tau}_2.\hat{\boldsymbol{h}} \neq \top \wedge \\ \left(\left(\begin{array}{l} \bar{\tau}_1.\hat{\boldsymbol{h}} \in H \wedge \\ \bar{\tau}_2.\hat{\boldsymbol{h}} \in H \end{array} \right) \Rightarrow \left[\begin{array}{ll} & \bar{\tau}_1 = \bar{\tau}_2 \\ \wedge & \bar{\tau}_1 < \top \\ \wedge & \forall w \in \mathbb{W} : \ (\bar{\tau}_1.\mathbf{\Pi}(w) = 1 \Rightarrow \\ & \qquad \bar{\tau}_3.\mathbf{\Pi}(w) = \bar{\tau}_4.\mathbf{\Pi}(w) = 1) \end{array} \right] \right) \end{array} \right)$$

Figure 9. Disjoint reachability property and disjoint reachability analysis.

only if no object $\bar{o}$ allocated at site h is reachable by one or more field dereferences from distinct objects $\bar{o}_1$ and $\bar{o}_2$ allocated at (not necessarily distinct) sites in H.

Before defining disjoint reachability analysis we need two definitions. We say $\bar{\tau}_1$ is *compatible* with $\bar{\tau}_2$, written $\bar{\tau}_1 \sim \bar{\tau}_2$, if they agree in all components where neither is $\top$:

$$\begin{array}{l} \bar{\tau}_1 \sim \bar{\tau}_2 \ \Leftrightarrow \\ \qquad (\bar{\tau}_1.\hat{\boldsymbol{h}} = \bar{\tau}_2.\hat{\boldsymbol{h}} \ \vee \ \bar{\tau}_1.\hat{\boldsymbol{h}} = \top \ \vee \ \bar{\tau}_2.\hat{\boldsymbol{h}} = \top) \wedge \\ \quad \forall w \in \mathbb{W} : \ (\bar{\tau}_1.\mathbf{\Pi}(w) = \bar{\tau}_2.\mathbf{\Pi}(w) \vee \bar{\tau}_1.\mathbf{\Pi}(w) = \top \vee \bar{\tau}_2.\mathbf{\Pi}(w) = \top) \end{array}$$

We say a type $\bar{\tau}$ is *less than* $\top$ if no component of $\bar{\tau}$ is $\top$:

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

To define disjoint reachability analysis, we define a transitive closure K^+ of K analogous to the transitive closure of C. Because the abstract effects in K may correspond to many concrete effects, the transitive closure of K is more involved than the transitive closure of C. Consider two effects $(\bar{\tau}_1 \trianglerighteq \bar{\tau}_2)$ and $(\bar{\tau}_3 \trianglerighteq \bar{\tau}_4)$. If $\bar{\tau}_2 \sim \bar{\tau}_3$ then $\bar{\tau}_2$ and $\bar{\tau}_3$ may abstract the same object and there is some transitive relationship between $\bar{\tau}_1$ and $\bar{\tau}_4$. Simple transitivity is sound in all but one case: If $\bar{\tau}_1.\mathbf{\Pi}(w) = 1 = \bar{\tau}_4.\mathbf{\Pi}(w)$ and either $\bar{\tau}_2.\mathbf{\Pi}(w) \neq 1$ or $\bar{\tau}_3.\mathbf{\Pi}(w) \neq 1$, then we cannot conclude that $\bar{\tau}_1$ and $\bar{\tau}_4$ are allocated in the same iteration of loop w. In this case it is sound to replace $\bar{\tau}_4.\mathbf{\Pi}(w)$ by $\top$, ensuring that there is no information about the relative allocation times with respect to loop w. We define $K^+ = \bigcup_{n \geq 1} K^n$, where K^n is:

DEFINITION 4.2. **(Closure of K)**

1. $K^1 = K$
2. If $(\bar{\tau}_1 \trianglerighteq \bar{\tau}_2) \in K^n$ and $(\bar{\tau}_3 \trianglerighteq \bar{\tau}_4) \in K$ then $(\bar{\tau}_1 \trianglerighteq \bar{\tau}_5) \in K^{n+1}$ provided $\bar{\tau}_2 \sim \bar{\tau}_3$ and
   (a) $\bar{\tau}_5.\hat{\boldsymbol{h}} = \bar{\tau}_4.\hat{\boldsymbol{h}}$
   (b) $\forall w \in \mathbb{W}$:

$$\bar{\tau}_5.\mathbf{\Pi}(w) = \begin{cases} \top & \text{if } \ \bar{\tau}_1.\mathbf{\Pi}(w) = 1 \ \wedge \ \bar{\tau}_4.\mathbf{\Pi}(w) = 1 \ \wedge \\ & \qquad (\bar{\tau}_2.\mathbf{\Pi}(w) \neq 1 \ \vee \ \bar{\tau}_3.\mathbf{\Pi}(w) \neq 1) \\ \bar{\tau}_4.\mathbf{\Pi}(w) & \text{otherwise} \end{cases}$$

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

LEMMA 4.3. If $C \preceq K$ then $C^n \preceq K^n$.

Proof. [sketch] By induction on n, and using the definition of $C \preceq K$ in Figure 7(b) and Definitions 4.1 and 4.2 of C^n and K^n, respectively. We omit the details. □

The second equation of Figure 9 defines disjoint reachability analysis. The idea is that the test $h \in DR_K(H)$ is a sufficient condition to show that $h \in DR_C(H)$, where $C \preceq K$. The quantification enforces that the test hold for every pair of effects $\bar{\tau}_1 \trianglerighteq \bar{\tau}_3$ and $\bar{\tau}_2 \trianglerighteq \bar{\tau}_4$ in K^+. Now, $\bar{\tau}_1$ and $\bar{\tau}_2$ correspond to $\bar{o}_1$ and $\bar{o}_2$ in the definition of the disjoint reachability property. The types $\bar{\tau}_3$ and $\bar{\tau}_4$ may abstract the same object if $\bar{\tau}_3 \sim \bar{\tau}_4$, so these two types play the role of $\bar{o}$ in the disjoint reachability property. Finally, for the pair of effects to be relevant to the disjoint reachability test, both $\bar{\tau}_3.\hat{\boldsymbol{h}}$ and $\bar{\tau}_4.\hat{\boldsymbol{h}}$ must be either h or $\top$.

If these four conditions are satisfied, then the pair of effects is relevant to the disjoint reachability test. Intuitively, the test must now check that whenever $\bar{\tau}_3$ and $\bar{\tau}_4$ are the same that $\bar{\tau}_1$ and $\bar{\tau}_2$ are also the same, which is done by the right-hand side of the first implication. If either $\bar{\tau}_1.\hat{\boldsymbol{h}}$ or $\bar{\tau}_2.\hat{\boldsymbol{h}}$ is $\top$ then the test fails; in this case $\bar{\tau}_1$ and $\bar{\tau}_2$ stand for objects allocated at any allocation site and we can never guarantee the objects they represent are equal. Otherwise, if either $\bar{\tau}_1.\hat{\boldsymbol{h}}$ and $\bar{\tau}_2.\hat{\boldsymbol{h}}$ are concrete allocation sites but not in the set H then this pair of effects does not affect whether the disjoint reachability property holds or not. Finally if $\bar{\tau}_1.\hat{\boldsymbol{h}}$ and $\bar{\tau}_2.\hat{\boldsymbol{h}}$ are both in H (we are now discussing the right-hand side of the second implication), then we require two things. First, $\bar{\tau}_1$ and $\bar{\tau}_2$ must correspond to the same concrete object, which is enforced by requiring the types to be equal $\bar{\tau}_1 = \bar{\tau}_2$ and that they have no top elements $\bar{\tau}_1 < \top$ (because, again, a top element in the loop vector would also allow the type to correspond to more than one runtime object and hence the concrete objects corresponding to the two types could not be shown to always be equal). The other condition, which is enforced by the last clause, is that whenever $\bar{\tau}_1.\mathbf{\Pi}(w) = 1$ (and therefore $\bar{\tau}_2.\mathbf{\Pi}(w) = 1$), we have $\bar{\tau}_3.\mathbf{\Pi}(w) = \bar{\tau}_4.\mathbf{\Pi}(w) = 1$, which guarantees that distinct $\bar{\tau}_1$ ($\bar{\tau}_2$) objects (i.e., allocated in different loop iterations) reach different $\bar{\tau}_3$ ($\bar{\tau}_4$) objects.

The following theorem states the soundness of disjoint reachability analysis.

THEOREM 4.4. **(Soundness of Disjoint Reachability Analysis)** If $C \preceq K$ and $h \in \mathit{DR}_K(H)$ then $h \in \mathit{DR}_C(H)$.

Proof. [sketch] From Lemma 4.3, using the definition of $C \preceq K$ in Figure 7(b) and the definitions of DR_K and DR_C in Figure 9. We omit the details. □

Recall from the end of Section 3 that the type and effect system derives the heap effect $K = \{\langle\langle 1\rangle, h_1\rangle \trianglerighteq \langle\langle 1\rangle, h_2\rangle\}$ for Example 2.1. In this simple example $K^+ = K$. The single effect says that every object allocated at h_2 is pointed to by an object allocated at h_1 in the same loop iteration; therefore $h_2 \in \mathit{DR}_K(\{h_1\})$ using the analysis in Figure 9. Now consider the nested loops in Example 3.5. Using statement (A), we have $K = \{\langle\langle 1, 0\rangle, h_1\rangle \trianglerighteq \langle\langle 1, 1\rangle, h_2\rangle$ and again $K^+ = K$. Because the right side of this effect has a 1 in every position where the left side has a 1 and the left side has no $\top$ elements, this effect says that every object allocated in the inner loop is pointed to by at most one object allocated in the outer loop; thus $h_2 \in \mathit{DR}_K(\{h_1\})$. Finally, using statement (B), we have $K = \{\langle\langle 1, 1\rangle, h_2\rangle \trianglerighteq \langle\langle 1, 0\rangle, h_1\rangle$ and $K^+ = K$. Because the left side of this effect has a 1 in a position where the right side has a 0, we can infer that multiple objects allocated in loop 2 at h_2 may point to an object allocated at h_1 outside of loop 2. Thus $h_1 \notin \mathit{DR}_K(\{h_2\})$.

5. Implementation

We have implemented conditional must not alias analysis using disjoint reachability analysis in Chord, a static race detection system for Java. In this section, we present the architecture of Chord and discuss extensions to the formalisms presented so far to handle a realistic language like Java.

Before beginning, we should explain the claim that our system is sound. Unfortunately, the term "sound" is used rather loosely in the current literature; generally speaking, works refer to their algorithm as sound if it is designed to be sound under a widely used set of assumptions, and we have adopted this convention. We ignore the effects of reflection, dynamic loading, and native methods; these assumptions are standard for Java. In this paper we also elide checking races on accesses in constructors and class initializers (a standard assumption for static race detection), but this limitation is to make comparison with experiments in previous work direct; Chord as described here can handle the analysis of constructors and class initializers with adequate precision.

5.1 Chord Architecture

Chord consists of two phases each of which comprises a series of stages. The first phase centers around a context-sensitive may alias analysis while the second is based on conditional must not alias analysis.

5.1.1 Context-Sensitive May Alias Analysis Phase

This phase consists of a series of five stages, called *original*, *reachable*, *aliasing*, *escaping*, and *may-happen-in-parallel*. These stages are similar, but not identical, to the stages used in our earlier unsound race detector [33].

All stages work on six-tuples of the form $(t_1, c_1, e_1, t_2, c_2, e_2)$ where e_1 and e_2 are expressions that may race (i.e., accesses to instance fields, static fields or array elements[2]) and t_1, c_1, t_2, c_2 are contexts in the sense of the context-sensitive may alias analysis around which this phase is centered. Our implementation uses k-object-sensitive may alias analysis [28, 32] and is parameterized by k. We find it necessary to distinguish two different kinds of contexts to gain adequate precision:

- c_1 is the *calling context* of the method m containing e_1. In k-object sensitivity, the calling context of a method m is the allocation context of m's `this` parameter. For instance, if $k = 1$ than the context is simply the allocation site of `this`. If m is a static method, which lacks the `this` parameter, the calling context is a distinguished context called ϵ [32].
- t_1 is the *thread context* of e_1. The thread context is the calling context of the starting method of the thread that executes e_1. In Java, this method is either the `main` method (in the case of the implicitly created main thread) or the `start()` method of class `java.lang.Thread` (in the case of any other thread). Thus, in k-object sensitivity, the thread context is either ϵ (since `main` is a static method) or the allocation context of the `start` method's `this` parameter.
- t_2 and c_2 are similarly the thread and calling contexts of e_2.

The may alias analysis phase begins with the *original* stage which considers every type-compatible pair of accesses e_1 and e_2 in every possible calling and thread context to be a possible race (where at least one of the accesses is a write). The goal of each of the next four stages in this phase is to rule out some pairs of accesses from the set of possible races. Very briefly, these stages work as follows. Let $\theta = (t_1, c_1, e_1, t_2, c_2, e_2)$ be a six-tuple as above:

- *Reachable:* Not all thread and calling context combinations represent actual executions. This stage retains θ as a possible race only if there is a path in the program's context-sensitive call graph from the entry point of the thread starting method in context t_1 to the program point of e_1 in context c_1 (and similarly for t_2, c_2, e_2). This stage is context-sensitive.
- *Aliasing:* There can be a race between two accesses only if they access the same memory location, that is, if the accesses may alias. This stage retains θ as a possible race only if e_1 in context c_1 may aliases e_2 in context c_2. This stage is also context-sensitive.
- *Escaping:* A memory location can be subject to a race only if it is thread-shared. This stage retains θ only if a thread-escape analysis shows that e_1 in context c_1 may escape the thread in which it was allocated (and similarly for c_2, e_2). The analysis is context- and flow-sensitive and more precise than the one used in our previous work [33].
- *May-happen-in-parallel:* Two accesses can race only if they can happen in parallel. This stage retains θ only if it is possible that e_1 and e_2 in their corresponding thread and calling contexts can execute at the same time. Unlike most other may-happen-in-parallel analyses, ours is oblivious to locks. The most important role of this flow- and context-sensitive stage is to analyze the thread spawning structure of the program. This stage is not present in our previous work; together with the improved thread-escape analysis, this stage enables us to eliminate the

[2] Our implementation does not separate array elements—all elements of an array are collapsed to a single abstract location.

hand annotations used in [33]—the system presented here requires no annotations.

5.1.2 Conditional Must Not Alias Analysis Phase

This phase consists of a series of three stages, called *global-lock*, *local-lock*, and *local-thread*, that rule out additional pairs of accesses from the set of possible races that survive the first phase. All three stages are based on the concept of conditional must not aliasing but differ operationally.

Let $\theta = (t_1, c_1, e_1, t_2, c_2, e_2)$ be a six-tuple as in the first phase:

- *Global-lock:* This stage rules out accesses guarded by global, uniquely named locks (see Section 1 for an example). Java programmers not only create such locks explicitly but also use such locks created by the virtual machine, for instance, by using `static synchronized` methods or by synchronizing on the `class` field of an object. This stage does not use disjoint reachability analysis since it does not need to track object reachability: it merely checks that some global lock is held along every path in the program's context-sensitive call graph from the entry point of the thread starting method in context t_1 to the program point of e_1 in context c_1, and that the same global lock is held similarly for t_2, c_2, e_2.
- *Local-lock:* This stage rules out accesses guarded by non-global locks. It determines whether along each pair of paths in the program's context-sensitive call graph from the entry points of the thread starting methods in contexts t_1 and t_2 to the program points of e_1 in context c_1 and e_2 in context c_2, respectively, some pair of locks e_3 and e_4 is held in contexts c_3 and c_4, respectively, such that e_3 and e_4 are prefixes of e_1 and e_2, respectively, and:

  $$(P(e_1, c_1) \cap P(e_2, c_2)) \subseteq \mathit{DR}_K(P(e_3, c_3) \cup P(e_4, c_4))$$

  where $P(e, c)$ denotes the points-to set of e in context c, K is the heap effect abstraction of the given program, and by "e is a prefix of e'", we mean e' is obtained by one or more field dereferences from e. The soundness of our disjoint reachability analysis coupled with the soundness of our points-to analysis guarantees that, whenever locks e_3 and e_4 are distinct, accesses e_1 and e_2 are also distinct and therefore race-free.
- *Local-thread:* This stage rules out thread-local accesses using the following variation on conditional must not alias analysis: if whenever two threads are distinct, then two expressions in those threads must refer to distinct locations, then those expressions cannot race. If neither t_1 nor t_2 is the ϵ context, then the thread starting method of both threads is the `start` method (as opposed to the `main` method), and we do the check as in the *local-lock* stage except that we substitute the `start` methods' `this` parameters for the locks (e_3 and e_4), and require the thread contexts t_1 and t_2 be c_3 and c_4, respectively. The same correctness argument applies: whenever threads e_3 and e_4 are distinct, accesses e_1 and e_2 are distinct and therefore race-free.

5.2 Extensions

Our presentation thus far has ignored some issues that are important in implementing our algorithm for a realistic programming language. One such issue is the treatment of field reads; consider the following example:

```
1.  x.f = y
2.  ... no writes to aliases of x.f ...
3.  z = x.f
```

According to Rule 14 of Figure 8 the information for `z` on line 3 is $\langle \lambda w.\top, \top)\rangle$; i.e., no useful information is known for `z`. Unfortunately, this rule is too coarse in practice, as there are situations similar to the one given above in realistic programs.

To improve the precision of the analysis we compute flow-sensitive must alias information for fields, e.g., after line 3 we want to know that `z` $=$ `y` (with `y`'s allocation site and loop information). The approach we use is a standard (but interprocedural) dataflow algorithm to track must alias facts on names of the form `x.f.g.h`...; there are similar algorithms in the literature [10].

This extension also introduces a new problem for a race detection algorithm that aims to be sound. Consider the read again on line 3 above. The conclusion that `z` $=$ `y` on line 3 is only valid if line 2 contains no writes to aliases of `x.f` and also no other thread writes an alias of `x.f`. It is not surprising that a flow-sensitive computation must reason about potential races, but it does lead to a recursively defined notion of race detection, as computing the set of races now depends on knowing the set of races to begin with. We use a standard iterative approach: initially we run race detection assuming the set of races R in the program is empty. Any discovered races are added to R and the entire algorithm is repeated; races in R are used to kill must alias dataflow facts where appropriate (so if `x.f` on line 3 is part of some race in R, then the assignment on line 3 yields $\langle \lambda w.\top, \top\rangle$ for `z`). The entire process repeats until R reaches a fixed point.

6. Experiments

In this section, we evaluate our implementation on a suite of four multi-threaded Java programs and provide a detailed comparison with our previous work [33].[3] Figure 10 shows, for each program in our benchmark suite, the number of application and library classes and lines of Java source code in the call graph computed using k-object-sensitive alias analysis, the value of k used, the total running time of Chord, and a brief description of the program.

Figure 11 shows the results of the first phase of our implementation. Columns (A) and (B) show the number of six-tuples $(t_1, c_1, e_1, t_2, c_2, e_2)$ and the number of pairs (e_1, e_2) in the set of possible races at the end of the first (*original*) and last (*may-happen-in-parallel*) stages, respectively, of this phase.[4] For comparison with our earlier race checker [33], we partition the latter six-tuples and pairs into *likely* and *unlikely* races: a six-tuple $(t_1, c_1, e_1, t_2, c_2, e_2)$ retained after the *may-happen-in-parallel* stage is an *unlikely* race if and only if either of the following holds:

- $t_1 = t_2$, in which case it is most likely a thread-local pair of accesses, or
- along every pair of paths in the program's context-sensitive call graph from the entry points of the thread starting methods in contexts t_1 and t_2 to the program points of e_1 and e_2 in contexts c_1 and c_2, respectively, some pair of locks e_3 and e_4 is held in contexts c_3 and c_4, respectively, such that $P(e_3, c_3) = P(e_4, c_4) = \{h\}$, that is, their points-to sets are singleton and equal, in which case it is most likely a pair of accesses guarded by a common lock.

Notice that both the above checks use may alias information to approximate must alias information. The approximation is effective for bug-finding in our earlier race checker: the user can choose to inspect only the small number of *likely* races that survive both the above checks. However, the approximation is also unsound and can result in false negatives buried in the large number of *unlikely* races

[3] We have not yet attempted the largest benchmarks used in our previous work [33], but we believe this algorithm can scale to those programs.

[4] The pairs count is the number of unique pairs of expressions appearing in all six-tuples—in many cases the same expressions have races in multiple contexts.

	app classes	lib classes	app LOC	lib LOC	k	time	brief description
`philo`	2	423	84	110,582	1	3m14s	Dining Philosophers Problem solver
`elevator`	5	425	531	111,147	1	5m43s	A real-time discrete event simulator
`tsp`	4	426	706	110,954	1	3m21s	Traveling Salesman Problem solver from ETH
`ftp`	118	478	21897	116,026	2	7m21s	Apache FTP Server

Figure 10. Benchmark characteristics.

that are uninspected. For instance, the first check ($t_1 = t_2$) prevents any race between a pair of threads spawned at the same allocation site from being reported as a *likely* race. Likewise, the second check does not do a disjoint reachability analysis style check to determine whether the same object accessed by e_1 and e_2 is reachable from distinct locks allocated at site h.

While our earlier race checker stops at the end of the first phase and presents the *likely* races to the user, our current race checker does not differentiate between *likely* and *unlikely* races and proceeds to perform the second phase on all possible races retained at the end of the first phase. The results of the second phase are shown in Figure 12. Column (C) shows the number of possible races retained after applying the *global-lock* stage to the results of the first phase, while Column (D) shows the number of possible races retained after the combined application of the *local-lock* and *local-thread* stages to the results of the *global-lock* stage. We present the results for these two stages combined since they are similar (they both employ disjoint reachability analysis), but Figure 13 shows the number of possible races retained after the *global-lock* stage that were proven to be guarded by a non-global lock (Column (F)) and proven to be thread-local (Column (G)) by these two stages. Column (H) shows the net effectiveness of our disjoint reachability analysis. Notice that the numbers of six-tuples in Columns (F) and (G) may not add up to those in Column (H) since certain six-tuples may be proven to be both, guarded by a non-global lock and thread-local. This may happen if application code creates and manipulates thread-local data using library classes like `java.util.Vector` and `java.io.*` that use synchronization extensively to ensure thread-safety of multi-threaded clients. Pairs (as opposed to six-tuples) of accesses are even more likely to figure in both columns, since reusable code is polymorphic in calling/thread context, with accesses in such code being guarded by a lock in one calling/thread context and thread-local in another.

The effectiveness of the *global-lock* stage is evident in the results for `tsp` and `ftp`. The `tsp` benchmark exclusively uses global locks: the number of pairs retained reduces from 494 to 140 in the *global-lock* stage, and from 140 to 30 in the *local-thread* stage, but no pairs are eliminated in the *local-lock* stage. The `ftp` benchmark also uses global locks heavily, though they are primarily in the library code exercised by this benchmark.

The effectiveness of disjoint reachability analysis is clear in the results for all four programs: 30 of the 33 pairs surviving after the *global-lock* stage for `philo` are proven thread-local (the remaining 3 are real races), while the majority of such surviving pairs for `elevator` and `tsp` are proven guarded by a non-global lock. Finally, many pairs are proven both guarded by a non-global lock and thread-local for `ftp` because this program creates and manipulates thread-local data using extensively synchronized library classes.

Finally, the effectiveness of the conditional must not alias phase is illustrated in the number of real races and false alarms reported in Column (D) in Figure 12, as compared to those resulting from the *likely* races produced by the first phase, shown in Column (E) in Figure 12. Our new algorithm found a total of 202 real races (counting pairs) in all four benchmarks; using likely races finds only 122, showing that our sound system can find significantly more races in real programs. The number of false alarms is generally larger in our new race checker; 115 total false positives of 317 reports is a 36% false positive rate. Most of the false positives in our new algorithm are the result of engineering shortcomings that are straightforward to remove with additional effort. Notice that the large numbers of *unlikely* races reported in Column (B) in Figure 11 that were left uninspected in our earlier work are almost completely eliminated. Significantly, 80 of these *unlikely* races turn out to be real races, a fact that is witnessed by the increase in the numbers of real races reported in Column (D) over those in Column (E) in Figure 12.

7. Related Work

In this section we very briefly survey the large literature on race detection (including dynamic techniques, static techniques, and work on checking atomicity) and the considerably smaller literature on problems related to conditional must not alias analysis. We begin by noting that our notion of loop vectors harkens back to the ideas of *iteration space* and *dependence distance* in work on vectorizing compilers. Loop vectors are points in the iteration space of the program, and our algorithm can be thought of as tracking dependent statements of distance 0 (i.e., in the same iteration). We are not aware of any deeper connections to the large literature on program parallelization; our focus is on linked data structures and locking while parallelizing compilers focus primarily on array references.

7.1 Dynamic Race Detection

To the extent that race detection is currently used in practice, practical race detectors are primarily dynamic. The most popular form of dynamic race detection is the *lockset algorithm* as exemplified by the Eraser tool [40]. Recent work has greatly improved the order-of-magnitude slowdowns of the original implementations [2, 43, 44] to the point that runtime overhead is generally much less than 50% [5]. Other recent work has combined approaches based on Lamport's happens-before relation [1, 7, 8, 11, 26, 31, 38, 41] with the lockset algorithm to mitigate the disadvantages of each [12, 23, 34, 35, 47].

Dynamic race detection suffers from the well-known problems of every dynamic analysis, namely that first it cannot be used with a partial, open program (such as a library) and second even when a closed program is available dynamic checking is dependent on having adequate input. Note that the lockset algorithm only depends on the set of locks held for each location and not the order in which locks are held, so a strength of the lockset algorithm is that it does not require a particular interleaving of thread executions to occur to detect races. But, for example, no dynamic race detector can find races in code that is not executed at all.

7.2 Static Race Detection

Static race detection offers the promise of being able to find races before programs are run or, in the case of libraries, even before they are fully written. Given the non-deterministic nature of races, the advantages of static analysis would seem greater for race detection than many other program verification problems. The history of static race detection has seen considerable theoretical progress using a variety of approaches (flow-insensitive type systems [3, 4, 15, 16, 22, 36, 39], flow-sensitive versions of the static lockset algorithm [6, 13, 42], or path sensitive model checkers [25, 37]) but

	(A) after *original* stage		(B) after *may-happen-in-parallel* stage					
			likely		unlikely		total	
	hexts	pairs	hexts	pairs	hexts	pairs	hexts	pairs
`philo`	123344	926	0	0	35	35	35	35
`elevator`	55680	662	0	0	320	305	320	305
`tsp`	298045	3379	4	4	490	490	494	494
`ftp`	97661557	55039	352	156	3733	1624	4085	1628

Figure 11. Results for phase 1: Context-sensitive may alias analysis.

	(C) after *global-lock* stage		(D) after (*local-lock* + *local-thread*) stages						(E) classification of *likely* races			
			real races		false alarms		total		real races		false alarms	
	hexts	pairs	hexts	pairs	hexts	pairs	hexts	pairs	hexts	pairs	hexts	pairs
`philo`	33	33	3	3	0	0	3	3	0	0	0	0
`elevator`	320	305	0	0	0	0	0	0	0	0	0	0
`tsp`	140	140	12	12	18	18	30	30	0	0	4	4
`ftp`	2780	1147	761	187	258	97	1019	284	297	122	55	34

Figure 12. Results for phase 2: Conditional must not alias analysis.

	(F) proven local-lock		(G) proven local-thread		(H) proven race-free	
	hexts	pairs	hexts	pairs	hexts	pairs
`philo`	30	30	0	0	30	30
`elevator`	52	37	268	268	320	305
`tsp`	0	0	110	110	110	110
`ftp`	1398	613	1156	577	1761	863

Figure 13. Effectiveness of disjoint reachability analysis.

until recently techniques were not known that scaled with sufficient precision to find large numbers of bugs in realistic programs.

Our own previous work, outlined in Section 5, scales reasonably well and finds many bugs in both open and closed programs [33]. Earlier work by Choi et al. [6] took the same basic approach, but found that scalability problems hampered effectiveness, probably because the idea of k-object-sensitive analysis did not exist then.

As discussed in Section 1, however, the final lock analysis phase of [33] is unsound, using a cheaper may alias analysis to decide when two abstract locks represent the same concrete lock, a problem requiring must alias analysis. Our desire to remove this limitation is the inspiration for this paper. Instead of modifying the original algorithm to use must alias information to show when two locks are the same, we instead have chosen to focus on the dual problem of showing when two locks are different. To carry out sound race detection, however, we must show that if two locks are different then they protect different locations, which leads directly to the more general definition of conditional must not alias analysis. Conditional must not alias analysis requires considerably deeper analysis of the heap and correspondingly of how the heap is constructed than traditional alias analyses—distinguishing multiple objects allocated at the same site is crucial, as is maintaining correlations between objects that are allocated and linked together.

7.3 Other Analysis Approaches

Disjoint reachability for data structures (e.g., proving that two lists constructed of separate elements are disjoint) is an old problem. Algorithms in this area range from flow-sensitive approximations of heap shape ([9] is an early example) to very powerful decision procedures [27, 30]. Our notion of disjoint reachability is less precise (e.g., it is flow-insensitive) but easier to scale to large programs and gives good results for race detection.

Ownership types express the idea that among all pointers to an object, one is often special in that it has more operations than other pointers. In the context of race detection, ownership can be used to prove encapsulation (that an object is the exclusive access to another object), which in turn can prove conditional must not aliasing facts: if two objects are distinct, any objects they encapsulate must be distinct. There are algorithms for inferring ownership [24] and encapsulation [21] and ownership types have been exploited in race detectors [3]. While encapsulation is sufficient to prove conditional must not aliasing, it is not necessary. Roughly speaking, ownership/encapsulation are properties of how an object is constructed, while conditional must not aliasing also considers the specific pointers through which an object is used; this more refined treatment can fully automatically check race freedom for objects that are not encapsulated and without ownership types.

Another related approach is that a *correlation* exists between two objects if they are used consistently together [36]. For race detection, correlation means a particular lock is always used to guard a particular location. *Correlation analysis* infers which locks always guard which locations. Our approach does not require locks and locations to be correlated; because each potentially racing pair of expressions is handled separately, different locks may be used to prove race freedom for the same expression in different pairs.

7.4 Atomicity Checking

Much recent work on verification of concurrent programs has focused on checking *atomicity* rather than race freedom [2, 14, 17–20, 39, 45, 46]. Atomicity is arguably a simpler and more natural property for programmers and experimental work suggests that programmers most often use locks to achieve atomicity. Regardless of whether future languages rely primarily on atomicity or locking, however, checking race freedom will be an important problem: existing concurrent languages will continue to use lock-based synchronization idioms heavily, and many algorithms for checking atomicity (in particular, those based on Lipton's theory of reduction [29]) reduce to checking race freedom.

8. Conclusions

We have introduced conditional must not aliasing and shown its application to static race detection. We have also presented disjoint reachability analysis, an object reachability analysis that is useful for computing conditional must not alias facts. We have implemented conditional must not alias analysis using disjoint reachability analysis in Chord, a static race checker for Java, and applied it to a suite of Java programs. The resulting system is fully automatic, reasonably efficient, and produces relatively few false positives.

Acknowledgments

We thank the anonymous POPL reviewers for insightful comments. This research was supported in part by NSF grants CCF-0430378 and CNS-0509558, a gift from Intel, and a Microsoft fellowship.

A. Proof of Type Preservation

We state a useful fact of environment abstraction (Figure 7(c)) that is needed in proving type preservation.

FACT A.1. **(Loopset Weakening)** If $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ and $W' \subseteq W$ then $W' \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$.

Proof. **[of Type Preservation]** By induction on the structure of the derivation of $s, W, \pi, \rho, \sigma \ \Downarrow\ \pi', \rho', \sigma', C$. There are 10 cases depending on which one of rules (1)–(10) in Figure 5 was used last in the derivation. For brevity, we only provide the proof for the two most interesting cases.

1. Rule (5). We have $s \equiv x.f = y$. There are two sub-cases depending upon whether $\rho(y)$ is null or non-null. We only prove the latter more interesting sub-case. We have:

   (a): $\rho(x) = \bar{o}_1$ and $\rho(y) = \bar{o}_2$ and

   (b): $s, W, \pi, \rho, \sigma \ \Downarrow\ \pi, \rho, \sigma[\bar{o}_1.f \mapsto \bar{o}_2], \{\bar{o}_1 \rhd \bar{o}_2\}$.

   From $s \equiv x.f = y$ and hypothesis $W, \Pi, \Gamma \vdash s : \Gamma', K$ of the lemma and rule (15) in Figure 8, we have:

   (c): $W, \Pi, \Gamma \ \vdash\ s : \Gamma, K$ where:

   $$K = \begin{cases} \{\bar{\tau}_1 \unrhd \bar{\tau}_2\} & \text{if } \Gamma(x) = \bar{\tau}_1 \text{ and } \Gamma(y) = \bar{\tau}_2 \\ \emptyset & \text{otherwise} \end{cases}$$

   We need to prove:

   (A): $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ and

   (B): $\{\bar{o}_1 \rhd \bar{o}_2\} \preceq K$.

   The proof of (A) is trivial since (A) is one of the hypotheses of the lemma. To prove (B), it suffices to prove (see Figure 7(b)):

   (d): $\exists(\bar{\tau}_1 \unrhd \bar{\tau}_2) \in K :\ (\bar{o}_1, \bar{o}_2) \propto (\bar{\tau}_1, \bar{\tau}_2)$.

   Proof of (d): From hypothesis $W \ \vdash\ (\pi, \rho) \ \preceq\ (\Pi, \Gamma)$ of the lemma, we have (see item (2) and item (3)(b) in Figure 7(c)):

   (e): $\forall z \in \mathbb{V} :\ \rho(z) \preceq \Gamma(z)$ and

   (f): $\forall w \in \mathbb{W} :\ \exists k \in \mathbb{N} :\ \forall z \in \mathbb{V} :\ ((\rho(z) = \bar{o} \ \wedge\ \Gamma(z) = \bar{\tau} \ \wedge\ \bar{\tau}.\mathbf{\Pi}(w) = 1) \ \Rightarrow\ \bar{o}.\boldsymbol{\pi}(w) = k)$.

   From (a) and (e), we have (see defn. of $o \preceq \tau$ in Figure 7(a)):

   (g): $\Gamma(x) = \bar{\tau}_1$ and $\Gamma(y) = \bar{\tau}_2$ and

   (h): $\bar{o}_1 \preceq \bar{\tau}_1$ and $\bar{o}_2 \preceq \bar{\tau}_2$.

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

   (i): $K = \{\bar{\tau}_1 \unrhd \bar{\tau}_2\}$.

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

   (j): $\forall w \in \mathbb{W} :\ \exists k \in \mathbb{N} :\ ((\bar{\tau}_1.\mathbf{\Pi}(w) = 1 \ \Rightarrow\ \bar{o}_1.\boldsymbol{\pi}(w) = k) \wedge (\bar{\tau}_2.\mathbf{\Pi}(w) = 1 \ \Rightarrow\ \bar{o}_2.\boldsymbol{\pi}(w) = k))$.

   From (h) and (j), we have (k): $(\bar{o}_1, \bar{o}_2) \propto (\bar{\tau}_1, \bar{\tau}_2)$ (see Figure 7(b)). From (i) and (k), we have (d).

2. Rule (10). We have $s \equiv \mathsf{while}^w\ (*)\ \mathsf{do}\ s'$ and

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

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

   From $s \equiv \mathsf{while}^w\ (*)\ \mathsf{do}\ s'$ and hypothesis $W, \Pi, \Gamma \vdash s : \Gamma', K$ of the lemma and rule (18) in Figure 8, we have:

   (c): $W, \Pi, \Gamma \ \vdash\ s : \Gamma, K$ and

   (d): $W \cup \{w\}, \Pi, \Gamma^{w+} \ \vdash\ s' : \Gamma, K$ and

   (e): $\Pi(w) \neq 0$.

   We need to prove:

   (A): $W \vdash (\pi'', \rho'') \preceq (\Pi, \Gamma)$ and

   (B): $(C_1 \cup C_2) \preceq K$.

   From hypothesis $W \vdash (\pi, \rho) \preceq (\Pi, \Gamma)$ of the lemma, we have (see Figure 7(c)):

   (f): $\forall w' \in W : \pi(w') \preceq \Pi(w')$ and

   (g): $\forall\, x \in \mathbb{V} : \rho(x) \preceq \Gamma(x)$ and

   (h): $\forall w' \in \mathbb{W} :\ \exists k' \in \mathbb{N} :$

   $(\Pi(w') = 1 \ \Rightarrow\ \pi(w') = k') \wedge$
   $(\forall x \in \mathbb{V} :\ ((\rho(x) = \bar{o} \ \wedge\ \Gamma(x) = \bar{\tau} \ \wedge\ \bar{\tau}.\mathbf{\Pi}(w') = 1) \ \Rightarrow\ \bar{o}.\boldsymbol{\pi}(w') = k'))$

   We will first prove:

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

   From Figure 7(c), this requires proving:

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

   (i.2): $\forall x \in \mathbb{V} :\ \rho(x) \preceq \Gamma^{w+}(x)$

   (i.3): $\forall w' \in \mathbb{W} :\ \exists k' \in \mathbb{N} :$

   $(\Pi(w') = 1 \ \Rightarrow\ (\pi[w \mapsto \pi(w) + 1])(w') = k') \wedge$
   $(\forall x \in \mathbb{V} :\ ((\rho(x) = \bar{o} \ \wedge\ \Gamma^{w+}(x) = \bar{\tau} \ \wedge\ \bar{\tau}.\mathbf{\Pi}(w') = 1) \ \Rightarrow\ \bar{o}.\boldsymbol{\pi}(w') = k'))$.

   Proof of (i.1): From (e), we have (j): $\Pi(w) = 1 \ \vee\ \Pi(w) = \top$. From $(\pi(w)+1) > 0$ and (j), we have (k): $(\pi(w)+1) \preceq \Pi(w)$ (see defn. of $i \preceq \hat{i}$ in Figure 7(a)). From (f) and (k), we have (i.1).

   Proof of (i.2): Immediate from (g) and defn. 3.2 of Γ^{w+} and the defn. of $o \preceq \tau$ in Figure 7(a).

   Proof of (i.3): It suffices to prove:

   (l): $\exists k \in \mathbb{N} :$

   $(\Pi(w) = 1 \ \Rightarrow\ (\pi[w \mapsto \pi(w) + 1])(w) = k) \wedge$
   $(\forall x \in \mathbb{V} :\ ((\rho(x) = \bar{o} \ \wedge\ \Gamma^{w+}(x) = \bar{\tau} \ \wedge\ \bar{\tau}.\mathbf{\Pi}(w) = 1) \ \Rightarrow\ \bar{o}.\boldsymbol{\pi}(w) = k))$.

   since we will have (i.3) from (h) and (l).

   Proof of (l): Choose $k = (\pi[w \mapsto \pi(w) + 1])(w)$ whence we have (m): $\Pi(w) = 1 \ \Rightarrow\ (\pi[w \mapsto \pi(w) + 1])(w) = k$. Also, from defn. 3.2 of Γ^{w+}, we have $\forall x \in \mathbb{V} :\ (\Gamma^{w+}(x) = \bar{\tau} \ \Rightarrow\ \bar{\tau}.\mathbf{\Pi}(w) \neq 1)$ whence we trivially have (n): $\forall x \in \mathbb{V} :\ ((\rho(x) = \bar{o} \ \wedge\ \Gamma^{w+}(x) = \bar{\tau} \ \wedge\ \bar{\tau}.\mathbf{\Pi}(w) = 1) \ \Rightarrow\ \bar{o}.\boldsymbol{\pi}(w) = k)$. From (m) and (n), we have (l).

   We now prove (A) and (B). From (a) and (d) and (i) and the induction hypothesis, we have (o): $W \cup \{w\} \ \vdash\ (\pi', \rho') \ \preceq\ (\Pi, \Gamma)$ and (p): $C_1 \ \preceq\ K$. From (o) and Fact A.1, we have (q): $W \ \vdash\ (\pi', \rho') \ \preceq\ (\Pi, \Gamma)$. From (b) and (c) and (q) and the induction hypothesis, we have (A) and (r): $C_2 \preceq K$. From (p) and (r), we have (B) (see Figure 7(b)).

□

References

[1] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting data races on weak memory systems. In *Proceedings of ISCA'91*, pages 234–243, 1991.

[2] R. Agarwal, A. Sasturkar, Wang L, and S. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of ASE'05*, pages 233–242, 2005.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of OOPSLA'02*, pages 211–230, 2002.

[4] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of OOPSLA'01*, pages 56–69, 2001.

[5] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of PLDI'02*, pages 258–269, 2002.

[6] J. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, 2001.

[7] J. Choi, B. Miller, and R. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM TOPLAS*, 13(4):491–530, 1991.

[8] M. Christiaens and K. Brosschere. TRaDe: A topological approach to on-the-fly race detection in Java programs. In *Proceedings of JVM'01*, pages 105–116, 2001.

[9] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

[10] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of PLDI'02*, pages 57–68, 2002.

[11] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of PPoPP'90*, pages 1–10, 1990.

[12] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of PADD'91*, pages 85–96, 1991.

[13] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of SOSP'03*, pages 237–252, 2003.

[14] C. Flanagan. Verifying commit-atomicity using model-checking. In *Proceedings of SPIN'04*, pages 252–266, 2004.

[15] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP'99*, pages 91–108, 1999.

[16] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of PLDI'00*, pages 219–232, 2000.

[17] C. Flanagan and S. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL'04*, pages 256–267, 2004.

[18] C. Flanagan, S. Freund, and M. Lifshin. Type inference for atomicity. In *Proceedings of TLDI'05*, pages 47–58, 2005.

[19] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI'03*, pages 338–349, 2003.

[20] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of TLDI'03*, pages 1–12, 2003.

[21] S. Ghemawat, K. Randall, and D. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of PLDI'00*, pages 334–344, 2000.

[22] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of TLDI'03*, pages 13–25, 2003.

[23] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of SPIN'00*, pages 331–342, 2000.

[24] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of PLDI'03*, pages 168–181, 2003.

[25] T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of PLDI'04*, pages 1–13, 2004.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[27] T. Lev-Ami, N. Immerman, T. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Proceedings of CADE'05*, pages 99–115, 2005.

[28] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of CC'06*, 2006.

[29] R. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18(12):717–721, 1975.

[30] S. McPeak and G. Necula. Data structure specifications via local equality axioms. In *Proceedings of CAV'05*, pages 476–490, 2005.

[31] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of SC'91*, pages 24–35, 1991.

[32] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of ISSTA'02*, pages 1–11, 2002.

[33] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of PLDI'06*, pages 308–319, 2006.

[34] R. O'Callahan and J. Choi. Hybrid dynamic data race detection. In *Proceedings of PPoPP'03*, pages 167–178, 2003.

[35] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of PPoPP'03*, pages 179–190, 2003.

[36] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of PLDI'06*, pages 320–331, 2006.

[37] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *Proceedings of PLDI'04*, pages 14–24, 2004.

[38] M. Ronsse and K. Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM TOCS*, 17(2):133–152, 1999.

[39] A. Sasturkar, R. Agarwal, L. Wang, and S. Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of PPoPP'05*, pages 83–94, 2005.

[40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of SOSP'97*, pages 27–37, 1997.

[41] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of PLDI'89*, pages 285–297, 1989.

[42] N. Sterling. WARLOCK - a static data race analysis tool. In *Proceedings of the Usenix Winter 1993 Technical Conference*, pages 97–106, 1993.

[43] C. von Praun and T. Gross. Object race detection. In *Proceedings of OOPSLA'01*, pages 70–82, 2001.

[44] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of PLDI'03*, pages 115–128, 2003.

[45] L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of PPoPP'05*, pages 61–71, 2005.

[46] L. Wang and S. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE TSE*, 32(2):93–110, 2006.

[47] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of SOSP'05*, pages 221–234, 2005.