Thread-Modular Shape Analysis

Alexey Gotsman
University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk

Josh Berdine
Microsoft Research
jjb@microsoft.com

Byron Cook
Microsoft Research
bycook@microsoft.com

Mooly Sagiv *
Tel-Aviv University
msagiv@post.tau.ac.il

Abstract

We present the first shape analysis for multithreaded programs that avoids the explicit enumeration of execution-interleavings. Our approach is to automatically infer a resource invariant associated with each lock that describes the part of the heap protected by the lock. This allows us to use a sequential shape analysis on each thread. We show that resource invariants of a certain class can be characterized as least fixed points and computed via repeated applications of shape analysis only on each individual thread. Based on this approach, we have implemented a thread-modular shape analysis tool and applied it to concurrent heap-manipulating code from Windows device drivers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Abstract interpretation, concurrent programming, shape analysis, static analysis

1. Introduction

The analysis of multithreaded programs is complicated in the context of heap-manipulation, in particular, in the presence of *deep heap update*, which occurs when linked data structures are altered after traversing some a priori unbounded distance. Flow-insensitive analysis for such programs is too imprecise. Hence, to date, the sound, accurate, and automatic analyses for these programs have implemented flow-sensitive analyses that rely on enumerating the interleavings of executions of threads in the program [26]. This leads to state-space explosion and unscalability.

Our goal in this paper is to create a shape analysis for programs with deep heap update that is scalable, sound, and accurate. We do so by constructing a shape analysis that avoids enumerating interleavings. Our approach is to infer a *resource invariant* [20, 19] associated with each lock that describes the part of the heap protected by the lock and has to be preserved by every thread. E.g., a resource invariant for a lock can state that the lock protects a cyclic doubly-linked list with a sentinel node pointed to by the variable *head*. For any given thread, the resource invariant restricts how other threads can interfere with it. If resource invariants are known, analyzing a multithreaded program does not require enumerating interleavings and can be done using a sequential shape analysis. The challenge is to infer the resource invariants.

A resource invariant describes two orthogonal kinds of information: it simultaneously carves out the part of the heap protected by the lock and defines the possible shapes that this part can have during program execution. Hence, informally, resource invariants for heap-manipulating programs are not least fixed points of any first-order equation. We show that, if we specify the borders of the part of the heap protected by a lock (i.e., the former kind of information), then we can characterize the shape of the part (i.e., the latter kind of information) as a least fixed point. This fixed point can be computed by repeatedly performing shape analysis on each individual thread, but not on the whole program, i.e., performing the analysis *thread-modularly*. The analysis is able to establish that the program being analyzed is memory-safe (i.e., it does not dereference heap cells that are not allocated), does not leak memory, and does not have data races (including races on heap cells).

We specify the borders of the part of the heap protected by a lock with two sets of program variables—the set of *entry points* and the set of *exit points*. The part of the heap protected by the lock is defined as everything that is reachable from entry points up to exit points. Therefore in the subheap carved out by a resource invariant, exit points are pointers that lead to parts of the global heap owned by others. Fortunately, for systems code (e.g., device drivers), we find that automatic tools [22, 25, 5] suffice for inferring the entry points[1]. In more complex cases the entry and exit points can be given by the user as annotations (or may be other analyses for them could be found).

Our approach can be rephrased as follows: we assume that the borders between parts of the heap owned by different threads or protected by different locks are *stable* in the sense that they are pointed to by fixed stack variables. These borders are not required to be immovable, and we do not preclude ownership transfer of heap cells between areas owned by different threads or locks. That is, while the stack variables which mark the borders are fixed, their values are mutable. So heap cells can move between the parts owned by different threads or protected by different locks.

Our contributions can be summarized as follows:

- We propose a framework for constructing thread-modular program analyses, which is particularly suitable for shape analyses due to the locality exhibited by the semantics of heap-manipulation (Section 3). Our framework is parametric in the sequential shape analysis domain and can be instantiated with different domains.

* A part of this work was done while visiting Microsoft Research, Cambridge, UK.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

[1] Moreover, the soundness of our analysis does not depend on the particular association of locks and entry points: the analysis can be used with any association.

$$
\begin{array}{lcl}
V & ::= & x, y, \ldots \\
E & ::= & \mathsf{null} \mid V \\
\Pi & ::= & \Pi \wedge \Pi \mid E = E \\
\Sigma & ::= & \Sigma * \Sigma \mid \mathsf{emp} \mid \mathsf{true} \mid E{\mapsto}\{\mathit{prev}\colon E, \mathit{next}\colon E\} \\
 & \mid & \mathsf{dll}(E, E, E, E) \\
\Phi & ::= & \exists \vec{V}.\Pi \wedge \Sigma
\end{array}
$$

Figure 2. A subset Φ of separation logic formulae

- We give a fixed-point characterization of a class of resource invariants for heap-manipulating programs that provides a way to compute them via a thread-modular fixed-point computation (Section 4). For programs that have resource invariants from this class, the precision of our thread-modular analysis depends only on the precision of the underlying (sequential) shape analysis.
- Based on the framework and on the fixed-point characterization we develop a thread-modular shape analysis (Section 5) and apply it to multithreaded heap-manipulating code from Windows device drivers (Section 6).

2. Illustrative example

The example program in a C-like language shown in Figure 1 represents a typical pattern occurring in systems code, such as Windows device drivers. In this case two concurrently executing threads are accessing the same cyclic doubly-linked list protected by a lock ℓ. The list is accessed via a sentinel head node pointed to by a variable h. In this example *thread1* adds nodes to the head of the list and *thread2* removes nodes from the head of the list. When applied to this code, the implementation of our analysis presented in Sections 5.2 and 6 establishes that the area of the heap protected by the lock—its resource invariant—has the shape of a cyclic doubly-linked list and that the program is memory-safe (i.e., it does not dereference heap cells that are not allocated), does not leak memory, and has no data races (including races on heap-cells).

The analysis uses an underlying sequential shape analysis, which is similar to the one presented in [9]. The abstract states and resource invariants in the analysis denote sets of stack-heap pairs and are represented by disjunctions of formulae in the subset of separation logic [23] defined in Figure 2. The informal meaning of the formulae is as follows:

- emp describes states where the heap is empty, with no allocated locations.
- $E{\mapsto}\{\mathit{prev}\colon E_1, \mathit{next}\colon E_2\}$ describes states where the heap contains a single allocated location E, with contents being a structure in which the fields *prev* and *next* are equal to E_1 and E_2. The values of the other fields in the structure are unspecified by this formula.
- $\Sigma_1 * \Sigma_2$ describes states where the heap is the union of two disjoint heaps (with no locations in common), one satisfying Σ_1 and the other satisfying Σ_2.
- $E_1 = E_2$ describes states where the stack gives E_1 and E_2 equal values.
- $\mathsf{dll}(E_1, E_2, E_3, E_4)$ is defined as the least predicate such that

$$
\begin{array}{c}
\mathsf{dll}(E_1, E_2, E_3, E_4) \Leftrightarrow \exists x.(E_1 = E_3 \wedge E_2 = E_4 \wedge \mathsf{emp}) \vee \\
(E_1{\mapsto}\{\mathit{prev}\colon E_2, \mathit{next}\colon x\} * \mathsf{dll}(x, E_1, E_3, E_4))
\end{array}
$$

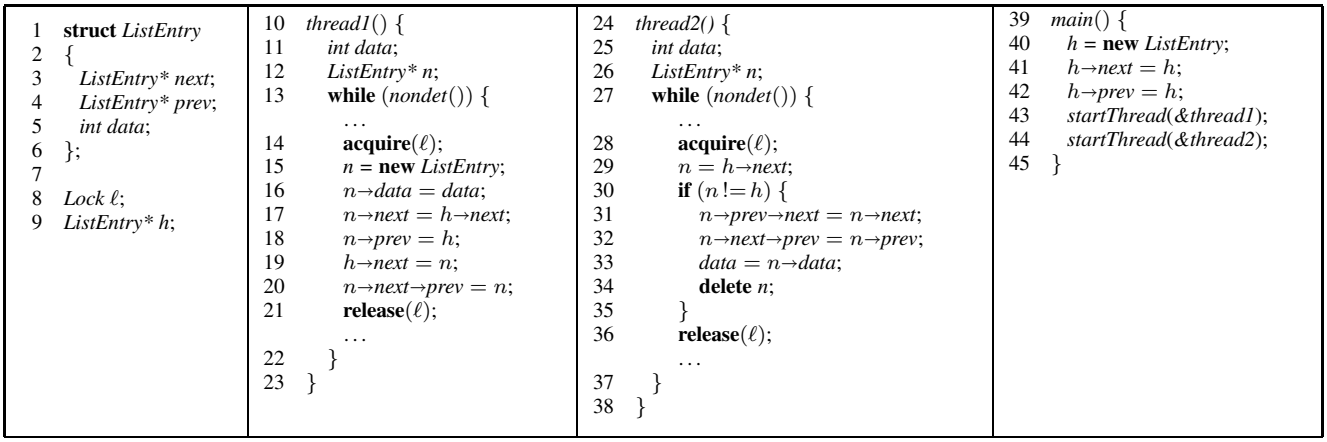
  and represents all of the states in which the heap has the shape of a (possibly empty) doubly-linked list, where E_1 is the address of the first node of the list, E_4 is the address of the last node, E_2 is the pointer in the *prev* field of the first node, and E_3 is the pointer in the *next* field of the last node.
- The meaning of propositional connectives, true and the existential quantifier is standard.

The analysis first uses a tool for analyzing correlations between locks and program variables, such as [22, 25, 5], to determine that the variable h is protected by the lock ℓ. The variable h becomes an *entry point* associated with the lock ℓ: the part of the heap protected by the lock is reachable from the entry point. There are no exit points associated with the lock ℓ in this example. The analysis is performed iteratively. On each iteration, we analyze the code of each thread and discover new shapes the part of the heap protected by each lock can have—new disjuncts in its resource invariant. On the next iteration each thread is re-analyzed taking the newly discovered disjuncts into account. This loop is performed until no new disjuncts in the resource invariant are discovered, i.e., until we reach a fixed point on the value of the resource invariant. Note that the particular order of the iteration is not important for the soundness of the analysis. In the example below we chose the order that is convenient to illustrate how the analysis works.

Executing the analysis. As a first step, we run the underlying sequential shape analysis on the *main* function to determine the initial approximation $I_0 = h{\mapsto}\{\mathit{prev}\colon h, \mathit{next}\colon h\}$ of the resource invariant associated with the lock ℓ. The initial states of the threads in this case are emp.

First iteration. We run the underlying sequential shape analysis on the code of *thread1* with the treatment for **acquire** and **release** commands described below. The analysis performs a fixed-point computation to determine invariants of all the loops in *thread1*. Suppose the analysis reaches line 14 with an abstract state s. Upon acquiring the lock ℓ the thread *gets ownership* of the part of the heap protected by the lock. We mirror this in the analysis by $*$-conjoining the current approximation I_0 of the resource invariant associated with the lock ℓ to the current state s yielding $s * I_0$. The analysis of the code in lines 15–20 starting from this state then gives us the state $s_1 = s * h{\mapsto}\{\mathit{prev}\colon n, \mathit{next}\colon n\} * n{\mapsto}\{\mathit{prev}\colon h, \mathit{next}\colon h\}$ at line 21. Upon releasing the lock ℓ the thread has to give up the ownership of the part of the heap protected by the lock. This means that the analysis has to partition the current heap s_1 into two parts, one of which becomes the local heap of the thread (the part of the heap that the thread owns) and the other is added as a new disjunct to the resource invariant. We compute the partitioning in the following way: the part of the heap reachable from the entry points associated with the lock ℓ becomes a new disjunct in the resource invariant and the rest of the heap becomes the local state of the thread. Intuitively, when a thread modifies pointers to a heap cell so that it becomes reachable from the entry points associated with a lock, the cell becomes protected by the lock and a part of its resource invariant. In this way, we discover a new disjunct $I_1 = \exists x.h{\mapsto}\{\mathit{prev}\colon x, \mathit{next}\colon x\} * x{\mapsto}\{\mathit{prev}\colon h, \mathit{next}\colon h\}$ in the resource invariant and a new state s reachable right after line 21. Note that since the variable n is not an entry point associated with the lock ℓ, we existentially quantify it in I_1. We keep running the fixed-point computation defined by the underlying sequential shape analysis starting from this state to discover the invariant of the loop in line 13 (as well as all other loops in the thread). The processing of lines 14 and 21 is the same as before, i.e., we use the same approximation I_0 of the resource invariant and get the same disjunct I_1. We stop when the underlying shape analysis reaches a fixed point. One new disjunct I_1 of the resource invariant has been discovered.

We now analyze the code of *thread2*. Whenever the analysis reaches line 28 with an abstract state q, we conjoin the current approximation I_1 of the resource invariant to the state q yielding



```
1  struct ListEntry
2  {
3    ListEntry* next;
4    ListEntry* prev;
5    int data;
6  };
7
8  Lock ℓ;
9  ListEntry* h;

10 thread1() {
11   int data;
12   ListEntry* n;
13   while (nondet()) {
       ...
14     acquire(ℓ);
15     n = new ListEntry;
16     n→data = data;
17     n→next = h→next;
18     n→prev = h;
19     h→next = n;
20     n→next→prev = n;
21     release(ℓ);
       ...
22   }
23 }

24 thread2() {
25   int data;
26   ListEntry* n;
27   while (nondet()) {
       ...
28     acquire(ℓ);
29     n = h→next;
30     if (n != h) {
31       n→prev→next = n→next;
32       n→next→prev = n→prev;
33       data = n→data;
34       delete n;
35     }
36     release(ℓ);
       ...
37   }
38 }

39 main() {
40   h = new ListEntry;
41   h→next = h;
42   h→prev = h;
43   startThread(&thread1);
44   startThread(&thread2);
45 }
```

Figure 1. Example program. *nondet*() represents non-deterministic choice.

$q * I_1$ and analyze the code in lines 29–35 starting from this state. This gives us the state $q_1 = q * h \mapsto \{prev: h, next: h\}$ at line 36. We again take the part of the heap reachable from h as a new disjunct in the resource invariant and let the rest of the heap be a new local state of the thread. In this case the new disjunct in the resource invariant is the same as the starting one I_0, so, no new disjuncts in the resource invariant are discovered.

Second iteration. On the previous iteration we found a new disjunct I_1 in the resource invariant associated with the lock ℓ. This means that whenever a thread acquires the lock ℓ, it can get ownership of a piece of heap with this new shape. To account for this in the analysis we now consider this possibility for all **acquire**(ℓ) commands in the program and perform the analysis on threads starting from the resulting new states. In *thread1* we obtain a new state $s * I_1$ at line 15. The analysis of the code in lines 15–20 in this case gives us the state $s_2 = \exists x.s * h \mapsto \{prev: x, next: n\} * n \mapsto \{prev: h, next: x\} * x \mapsto \{prev: n, next: h\}$. at line 21. Again, the part of the heap reachable from h forms a new disjunct in the resource invariant. To ensure convergence we abstract it before saving: the abstraction procedure similar to the one presented in [9] abstracts the heap that has two cells n and x connected in a doubly-linked list to a general doubly-linked list giving us a new disjunct in the resource invariant: $I_2 = \exists x, y.h \mapsto \{prev: x, next: y\} * \mathsf{dll}(y, h, h, x)$. A similar procedure for *thread2* again gives us the state q_1 at line 36. No new disjuncts in resource invariants are discovered while analyzing this thread.

Third iteration. We propagate the newly discovered disjunct I_2 of the resource invariant to **acquire** commands. The new state $s * I_2$ at line 15 gives rise to the state $s_3 = \exists x, y.s * h \mapsto \{prev: x, next: n\} * n \mapsto \{prev: h, next: y\} * \mathsf{dll}(y, n, h, x)$ at line 21. Partitioning it into the part reachable from h and the part unreachable from h and abstracting the latter gives us again the resource invariant I_2 and the state s. Propagating the new disjunct in the resource invariant to line 28 yields the state $q_2 = \exists x, y.q * h \mapsto \{prev: x, next: y\} * \mathsf{dll}(y, h, h, x)$ at line 36. Partitioning this state again does not result in new disjuncts in the resource invariant being discovered.

No new disjuncts in resource invariants were discovered on this iteration, hence, we have reached a fixed point. The resource invariant for the lock ℓ computed by the analysis is $I_0 \vee I_1 \vee I_2$. Furthermore, the program is memory-safe. □

3. Thread-modular shape analyses

In this section we present a general framework for constructing thread-modular shape analyses based on the inference of resource invariants associated with each lock that describe the part of the program state protected by the lock. Abstracting from particular domains used in shape analyses we formulate it in general lattice-theoretic terms. We use the framework first in Section 4 to give a fixed-point characterization of resource invariants for heap-manipulating programs and then in Section 5 to implement a thread-modular shape analysis.

3.1 Preliminaries

We consider concurrent programs consisting of a bounded number of threads that use a bounded number of non-aliased locks for synchronization. Each thread is represented by its control-flow graph (CFG). For any nodes v_1 and v_2 in a CFG there are three types of edges that can connect them:

- (v_1, C, v_2), where C is an element of a fixed set $\mathcal{C}$ of sequential commands;
- $(v_1, \textbf{acquire}(\ell), v_2)$ corresponding to acquiring the lock ℓ;
- $(v_1, \textbf{release}(\ell), v_2)$ corresponding to releasing the lock ℓ.

Consider a program with m threads $P_1, \ldots, P_m$ in which P_i is represented by a CFG with the set of nodes N_i and the set of edges E_i. Let $\mathcal{L} = \{\ell_1, \ldots, \ell_n\}$ be the set of locks used in the program. Let $N = \bigcup_{i=1}^{m} N_i$, $E = \bigcup_{i=1}^{m} E_i$, and start_i be the start node of thread i. Without loss of generality we assume that there are no edges in the CFG of the program leading to a start node. We call a tuple $(v_1, \ldots, v_m)$, where $v_i \in N_i$ a location. We now define a collecting interleaving semantics for the program.

In this paper by a *domain* D we understand a join-semilattice $(D, \sqsubseteq, \bigsqcup, \bot, \top)$ with a bottom element $\bot$. We assume given a domain D representing sets of states of the program and a set of monotone concrete transfer functions $f_C : D \to D$ representing the semantics of sequential commands $C \in \mathcal{C}$. The function f_C maps pre-states to states obtained by executing the command C from a pre-state.

Programs in our semantics denote mappings from locations and sets of locks held by each thread to elements of the domain D. Formally, programs denote elements of the domain $\widehat{D} = (N_1 \times \ldots \times N_m) \to ((\mathcal{P}(\mathcal{L}))^m \to D)$ ordered by the pointwise extension of $\sqsubseteq$. We call an element from $(\mathcal{P}(\mathcal{L}))^m$ a lockset and say that a lockset $(L_1, \ldots, L_m)$ is *admissible* if the sets of

$F(q) = q'$ where

- $q'(\mathsf{start}_1, \ldots, \mathsf{start}_m, \emptyset, \ldots, \emptyset) = \mathsf{pre}$;
- $q'(v_1, \ldots, v_m, L_1, \ldots, L_m) = \bigsqcup_{j=1}^{m} \bigsqcup_{(v_j^0, C, v_j) \in E} g_C^j(q(v_1, \ldots, v_j^0, \ldots, v_m), L_1, \ldots, L_m)$, if $(L_1, \ldots, L_m)$ is admissible, where

$$g_C^j(s, L_1, \ldots, L_m) = \begin{cases} f_C(s(L_1, \ldots, L_m)), & \text{if } C \text{ is a sequential command;} \\ s(L_1, \ldots, L_j \setminus \{\ell_i\}, \ldots, L_m) & \text{if } C \text{ is } \textbf{acquire}(\ell_i) \text{ and } \ell_i \in L_j; \\ s(L_1, \ldots, L_j \cup \{\ell_i\}, \ldots, L_m) \sqcup s(L_1, \ldots, L_j, \ldots, L_m) & \text{if } C \text{ is } \textbf{release}(\ell_i) \text{ and } \ell_i \notin L_j; \\ \bot, & \text{otherwise.} \end{cases}$$

Figure 3. The functional F defining the concrete collecting semantics for a multithreaded program

locks held by different threads are disjoint, i.e., for each i and j such that $i \neq j$ it is the case that $L_i \cap L_j = \emptyset$. For $q \in \widehat{D}$, $q(v_1, \ldots, v_m, L_1, \ldots, L_m)$, denotes the set of reachable states of the program at the location $(v_1, \ldots, v_m)$ such that the lockset held by threads is $(L_1, \ldots, L_m)$. The reason for having a lockset as an argument of q is that locking does not have to be lexically scoped, hence, reachable states at each location may have different locksets.

We assume a given $\mathsf{pre} \in D$ representing the initial states of the program. The semantics of the program is defined using the functional $F : \widehat{D} \to \widehat{D}$ that takes a function $q \in \widehat{D}$ and maps it to a function $q' \in \widehat{D}$ following the rules shown in Figure 3. Since the transfer functions f_C are monotone, by Tarski's fixed point theorem the functional F has least fixed point $\mathsf{lfp}(F)$, which represents the denotation of the program.

Consider the second equation in Figure 3. According to it, to compute the state at any location (except for the initial one) we consider all the edges in the CFG that can be taken by any single thread that lead to this location (hence, the semantics is based on interleaving) and take the join of the application of the function g for each of these edges (hence, the semantics is collecting) to the state at the source node. Note that here we use partial application of the function q. The function g defines the semantics of the statement at each edge of the CFG. For a sequential command it applies the transfer function for this command. Post-states of **acquire**(ℓ_i) for a lockset in which the thread P_j holds the lock ℓ_i are the same as pre-states in which the thread P_j does not hold the lock ℓ_i. Hence, acquiring a lock by a thread corresponds to adding this lock to the lockset of the thread. For the **acquire**(ℓ_i) command the function g filters out the pre-states in which the thread P_j holds the lock ℓ_i. Hence, a thread locking the same lock twice deadlocks. Post-states of **release**(ℓ_i) for a lockset in which the thread P_j does not hold the lock ℓ_i contain pre-states in which the thread P_j does not hold the lock ℓ_i as well as pre-states in which it does. Hence, releasing a lock by a thread corresponds to removing it from the lockset of the thread if it is there, and to a no-op if it is not there. This semantics of releasing a lock corresponds to treating locks as binary semaphores.

3.2 Abstract interpretation with state separation

As can be seen from the illustrative example in Section 2, in our thread-modular shape analysis we have to split abstract heaps into disjoint parts. For this to be possible the concrete and abstract domains have to have a separated structure that allows for performing such splittings. In this section we specialize the conventional notion of abstract interpretation [7] for the case when the domains have such a structure. In the subsequent sections we use this specialization as a foundation for designing thread-modular shape analyses.

DEFINITION 1 (Separation domain). *A separation domain is a domain $(D, \sqsubseteq, \bigsqcup, \bot, \top, e, *)$ equipped with an operation of separate combination $* : (D \times D) \to D$ such that $(D, \sqsubseteq, e, *)$ is a partially-ordered commutative monoid, i.e.:*

$$\begin{array}{l|l} \text{Values} = \{\ldots, -1, 0, 1, \ldots\} & \text{Locs} = \{1, 2, \ldots\} \\ \text{Vars} = \{x, y, \ldots\} & \text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} \text{Values} \\ \text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \text{Values} & \text{States} = (\text{Stacks} \times \text{Heaps}) \cup \{\top\} \\ D = \mathcal{P}(\text{States}) & \end{array}$$

Figure 4. Example of a separation domain

- $*$ *is associative and commutative:*

$$\forall u, v, w \in D.\ u * (v * w) = (u * v) * w;$$
$$\forall u, v \in D.\ u * v = v * u;$$

- $*$ *has the unit* e*:* $\forall u \in D.\ u * e = u$*;*
- $*$ *is monotone:* $\forall u_1, u_2, v \in D.\ u_1 \sqsubseteq u_2 \Rightarrow u_1 * v \sqsubseteq u_2 * v$.

We use a slight variation on the following instance of a separation domain in the further sections to design a thread-modular shape analysis.

Example. Figure 4 defines a separation domain D for the concrete semantics of heap-manipulating programs [27]. A state of the program is a stack-heap pair or a special error state $\top$. A stack is a finite partial function from variables to values, a heap is a finite partial function from locations to values. The domain consists of sets of states of the program. We identify all the sets of states containing $\top$ and denote such elements of the domain simply with $\top$. The order in the domain D is subset inclusion with $\top$ being the topmost element. This is essentially equivalent to using a topped powerset. However, in the further sections the formulation we use here allows us to simplify certain definitions.

In this paper we use the following notation for partial functions: $f(x)\!\downarrow$ means that the function f is defined on x, $f(x)\!\uparrow$ means that the function f is undefined on x, and $\mathsf{dom}(f)$ denotes the set of arguments on which the function f is defined. We denote with $f[x : y]$ the function that has the same value as f everywhere, except for x, where it has the value y (even if $f(x)\!\uparrow$). $f \uplus g$ is the union of the disjoint partial functions f and g. It is undefined if $\mathsf{dom}(f) \cap \mathsf{dom}(g) \neq \emptyset$. We denote with $f|_d$ the function identical to f except for its domain has been restricted to the set d.

We define the operation of separate combination on the domain D in the following way: for $s_1, s_2 \in \text{States}$

$$s_1 * s_2 = \{(t_1 \uplus t_2, h_1 \uplus h_2) \mid (t_1, h_1) \in s_1 \wedge (t_2, h_2) \in s_2\},$$
$$s_1 * \top = \top * s_2 = \top.$$

The unit element with respect to this operation is a singleton set containing a pair of everywhere undefined functions. A reader familiar with separation logic [23] can immediately notice that the definition of $*$ we gave here corresponds to the model of the separating conjunction from separation logic in the case when variables are treated as resources [21]. □

As can be seen from the example above, in this paper we use the topmost element of a separation domain to indicate a potential error. In this case the $*$ operation should also satisfy the following requirement:

$$\forall u, v \in D.\ u * v = \top \Rightarrow u = \top \vee v = \top. \tag{1}$$

That is, $*$ does not produce the error state unless one of its arguments is the error state.

For a program analysis to benefit from the structure present in a separation domain, transfer functions defining the concrete semantics of sequential program statements have to behave in a local way with respect to this structure. The following definition formalizes this condition.

DEFINITION 2 (Local function). *A function $f : D \to D$ defined over a separation domain $(D, \sqsubseteq, \bigsqcup, \bot, \top, e, *)$ is local if for all $u, v \in D$ it is the case that*

$$f(u * v) \sqsubseteq f(u) * v. \tag{2}$$

For the separation domains we consider in this paper, intuitively, if f is the meaning of a command C, this condition requires that if executing C from a state in $u * v$ results in an error $f(u * v) = \top$, then executing C from a smaller state in u also produces an error: $\top \sqsubseteq f(u) * v$ implies $f(u) = \top$ by (1). Furthermore, if executing C from a state in u does not produce an error, then executing C from a larger state, in $u * v$, has the same effect and leaves v unchanged: $f(u * v) = f(u) * v$.

The construction of thread-modular shape analyses is possible due the fact that concrete transfer functions for all standard heap-manipulating commands are local as illustrated by the following example.

Example. Consider the domain D from the previous example and a transfer function $f : D \to D$ corresponding to the command that stores the value of the variable y at the address equal to the value of the variable x, in C syntax "$*x = y$". We first define a function $f : \text{States} \to D$. For $t \in \text{Stacks}$ and $h \in \text{Heaps}$

$$f(t, h) = \begin{cases} (t, h[t(x) : t(y)]), & \text{if } t(x)\downarrow, t(y)\downarrow, h(t(x))\downarrow; \\ \top, & \text{otherwise,} \end{cases}$$

We let $f(\top) = \top$ and lift f to D pointwise.

The function f is local—when run on a piece of state it either produces the same result as when run on the extended state or it produces $\top$. □

Consider a concurrent programming language from the class defined in Section 3.1. In Section 3.3 we show how, given an analysis for the underlying sequential language, we can construct a thread-modular analysis for the concurrent language. More precisely, we assume given:

- a concrete separation domain $(D, \sqsubseteq, \bigsqcup, \bot, \top, e, *)$ representing sets of concrete states of the program;
- an abstract separation domain $(D^\sharp, \sqsubseteq, \bigsqcup, \bot, \top, e^\sharp, \sharp)$ representing sets of abstract states of the program;
- a monotone concretization function $\gamma : D^\sharp \to D$;
- monotone concrete transfer functions $f_C : D \to D$ defining the concrete semantics of sequential commands $C \in \mathcal{C}$;
- abstract transfer functions $f^\sharp_C : D^\sharp \to D^\sharp$ defining the abstract semantics of sequential commands $C \in \mathcal{C}$.

We assume further that:

- concrete transfer functions f_C are local;

$F^\sharp(Q, I) = (Q', I')$ where

- $Q'(\mathsf{start}_i, \emptyset) = \mathsf{Pre}_i(\mathsf{pre}^\sharp)$;
- $Q'(v, L) = \bigsqcup\limits_{(v^0, C, v) \in E} g^\sharp_C(Q(v^0), L)$ for every node $v \in N$, where $g^\sharp_C(s, L) =$
$$\begin{cases} f^\sharp_C(s(L)), & \text{if } C \text{ is a sequential command;} \\ s(L \setminus \{\ell_i\}) \sharp I_i, & \text{if } C \text{ is } \textbf{acquire}(\ell_i) \text{ and } \ell_i \in L; \\ \mathsf{Local}_i(s(L \cup \{\ell_i\})) \sqcup s(L), & \text{if } C \text{ is } \textbf{release}(\ell_i) \text{ and } \ell_i \notin L; \\ \bot, & \text{otherwise;} \end{cases}$$
- $I'_i = \mathsf{Init}_i(\mathsf{pre}^\sharp) \sqcup \bigsqcup\limits_{\substack{(v^0, \textbf{release}(\ell_i), v) \in E, \\ \{\ell_i\} \cap L \neq \emptyset}} \mathsf{Frame}_i(Q(v^0, L))$

  for each lock ℓ_i.

Figure 5. The functional $F^\sharp$ defining a thread-modular analysis. Functions Local_i and Frame_i define a heuristic that decides how to split the state upon releasing a lock. Pre_i and Init_i decide how to split the initial abstract state $\mathsf{pre}^\sharp$ between threads and locks.

- γ is a homomorphism between the monoids in the abstract and concrete separation domains:

$$\forall u, v \in D^\sharp.\ \gamma(u \sharp v) = \gamma(u) * \gamma(v); \tag{3}$$

- abstract transfer functions over-approximate the concrete ones:

$$\forall u \in D^\sharp, C \in \mathcal{C}.\ f_C(\gamma(u)) \sqsubseteq \gamma(f^\sharp_C(u)). \tag{4}$$

Note that we use the same symbols for the order, bottom and top elements, and the join operator for both domains when it does not cause confusion. Note also that we do not require that the abstract transfer functions be local or monotone.

3.3 Constructing thread-modular shape analyses

We now define a thread-modular analysis on a multithreaded program. The main idea of the analysis is to infer the part of the state protected by each lock—its resource invariant. Resource invariants are computed incrementally during the analysis, therefore, for each lock ℓ_i the analysis maintains the current approximation $I_i \in D^\sharp$ of a corresponding resource invariant. In addition, for every node v in the CFG and every set of locks L the analysis maintains the part of the state $Q(v, L) \in D^\sharp$ owned by the thread at the node v in the case when the set of locks held by the thread is L—its local state. Formally, the analysis operates on the domain $\widehat{D}^\sharp = (N \to (\mathcal{P}(\mathcal{L}) \to D^\sharp)) \times (D^\sharp)^n$ ordered by the pointwise extension of the abstract order $\sqsubseteq$.

We denote with $\circledsharp$ the iterated version of $\sharp$: $\circledsharp_{i=1}^k x_i = x_1 \sharp \ldots \sharp x_k$.

The thread-modular analysis is defined using the functional $F^\sharp : \widehat{D}^\sharp \to \widehat{D}^\sharp$ that takes a tuple (Q, I) and produces a tuple (Q', I') as shown in Figure 5. The analysis receives as input an abstract initial state of the program $\mathsf{pre}^\sharp \in D^\sharp$ such that

$$\mathsf{pre} \sqsubseteq \gamma(\mathsf{pre}^\sharp) \tag{5}$$

and is parameterized with the following functions:

- $\mathsf{Local}_i : D^\sharp \to D^\sharp$ and $\mathsf{Frame}_i : D^\sharp \to D^\sharp$ for each $i = 1..n$ such that for all $s \in D^\sharp$

$$\gamma(s) \sqsubseteq \gamma(\mathsf{Local}_i(s) \sharp \mathsf{Frame}_i(s)); \tag{6}$$

- $\mathsf{Pre}_i : D^\sharp \to D^\sharp$ for each $i = 1..m$ and $\mathsf{Init}_i : D^\sharp \to D^\sharp$ for each $i = 1..n$ such that for all $s \in D^\sharp$

$$\gamma(s) \sqsubseteq \gamma\left(\left(\bigsharp_{i=1}^{m} \mathsf{Pre}_i(s)\right) \sharp \left(\bigsharp_{i=1}^{n} \mathsf{Init}_i(s)\right)\right). \qquad (7)$$

Pre_i and Init_i determine the initial splitting of abstract state $\mathsf{pre}^\sharp$ between threads and locks. Pre_i and Init_i map the abstract initial state of the program $\mathsf{pre}^\sharp$ to the abstract initial state of thread i, respectively, the initial approximation of the resource invariant I_i. The condition (7) ensures that, when recombined, the results are an over-approximation of the abstract initial state.

The interesting part of the analysis concerns the treatment of acquiring and releasing locks. When a thread acquires a lock ℓ_i, it obtains the current approximation of the corresponding resource invariant—the current approximation of the resource invariant is $\sharp$-conjoined with the current local state of the thread to yield a new local state.

When a thread releases the lock ℓ_i, its current local state is partitioned into two parts, one of which returns to the resource invariant and the other one stays with the thread. The functions Local_i and Frame_i determine this splitting. The function Local_i determines the part of the state that becomes the local state of the thread and the function Frame_i the part that goes to the resource invariant. The condition (6) ensures that the combination of the parts of the splitting over-approximates the given abstract state. Note that the treatment of locksets in processing **acquire** or **release** commands in Figure 5 mimics the one in the concrete semantics (Figure 3).

A computation of a fixed point of the functional $F^\sharp$ would analyze each thread accumulating possible values of resource invariants during the analysis. Each time a new possible value of a resource invariant associated with a lock is discovered, it would have to be propagated to every **acquire** command for the lock. Hence, each thread is analyzed repeatedly, but separately, without exploring the set of interleavings. In this sense the analysis defined by $F^\sharp$ is thread-modular. Note also that after the analysis splits the state at a **release** command, it loses correlations between the parts of the state that become local states of the thread and the parts that go to the resource invariant. This loss of precision is similar to the one observed in thread-modular model checking [11].

We are now in a position to state and prove the soundness of the analysis defined by the functional $F^\sharp$. The following theorem says that reachable states at the location $(v_1, \ldots, v_m)$ such that the lockset held by threads is $(L_1, \ldots, L_m)$ are over-approximated by the combination of the local states of all threads along with resource invariants associated with the locks that are not in the lockset. The conditions listed at the end of Section 3.2 pinpoint the sufficient requirements that the underlying sequential analysis has to satisfy for the thread-modular analysis to be sound and are used in the proof of the theorem.

THEOREM 1 (Soundness). *Let q be least fixed point of the functional F defined in Figure 3 and (Q, I) be a fixed point of the functional $F^\sharp$ defined in Figure 5. Then for each location $(v_1, \ldots, v_m)$ and admissible lockset $(L_1, \ldots, L_m)$*

$$q(v_1, \ldots, v_m, L_1, \ldots, L_n) \sqsubseteq \gamma\left(\left(\bigsharp_{i=1}^{m} Q(v_i, L_i)\right) \sharp \left(\bigsharp_{\ell_i \notin L} I_i\right)\right)$$

where $L = L_1 \cup \ldots \cup L_m$.

The proof appears in Appendix A.

Note that although we have assumed a fixed number of threads, from the definition of the functional $F^\sharp$ it follows that the results of the analysis are sound for an unbounded number of copies of these threads provided they have the same initial states as the original ones. Formally, if we add an extra copy of a thread P_i into the program and change the concrete and the abstract initial states so that the condition (7) is still satisfied (which corresponds to adding an extra piece of state to the initial state of the program to form a precondition for a newly added thread), then the results of the analysis for the new program with the Q for the new thread equal to the Q for P_i still form a fixed point of the new functional $F^\sharp$ and, hence, over-approximate the concrete semantics.

Similar versions of the specialization of abstract interpretation presented in Section 3.2 were developed independently and can also be used as a basis for scaling up other static analysis algorithms even for programs without multithreading and dynamically allocated memory (see [13] for more information). For example, they are applicable in interprocedural analysis to split off the (abstract) state of the called procedure from the abstract state of the caller [24, 12].

V	::=	$x, y, \ldots$	variables
E	::=	$\mathsf{null} \mid V$	expressions
G	::=	$E == E \mid E != E$	branch guards
C	::=	$V = E \mid V = V{\to}\mathit{next}$	sequential commands
		$\mid\ V{\to}\mathit{next} = E$	
		$\mid\ V =$ **new** $\mid$ **delete** V	
		$\mid$ **assume**(G)	

Figure 6. Sequential commands for a heap-manipulating concurrent programming language

4. Resource invariants as least fixed points

In this section we show that if the borders of the part of the heap protected by a lock are specified, then the resource invariant describing this part can be defined as least fixed point of a first-order equation. For programs that have resource invariants from the class we define here the precision of our thread-modular analysis depends only on the precision of the underlying (sequential) shape analysis.

To show how to design thread-modular shape analyses we abstract from the particular shape domain used and construct an idealistic shape analysis that operates on concrete states. We do this by instantiating the framework of Section 3.3 with the same concrete and abstract domains. In Section 5 this instantiation serves as a template for designing thread-modular shape analyses. The instantiation provides a fixed-point characterization of a class of resource invariants and can be seen as defining a non-standard concrete semantics. Theorem 1 ensures its adequacy with respect to the standard collecting interleaving semantics.

4.1 Programming language and concrete semantics

Programming language. We consider a concurrent heap-manipulating programming language from the class of concurrent CFG-based languages defined in Section 3.1 with sequential commands shown in Figure 6. To simplify presentation we assume that each structure stored in the heap has only one field *next*. The development carried out in this section generalizes easily to the case of structures with multiple fields. The meaning of commands is standard. **assume**(G) acts as a filter on the state space of programs—G is assumed to be true after **assume** is executed. It is used to replace conditional expressions in **while** and **if** statements while translating programs to CFGs.

Consider a program in the language introduced above consisting of threads $P_1, \ldots, P_m$ (represented by their CFGs) with locks $\ell_1, \ldots, \ell_n$ and a precondition pre.

The domain of states. We now define a concrete semantics and a corresponding concrete separation domain $(D, \sqsubseteq, \bigsqcup, \bot, \top, e, *)$

$$\begin{array}{l|l} \text{Values} = \{\ldots,-1,0,1,\ldots\} & \text{Locs} = \{1,2,\ldots\} \\ \text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} \text{Values} & \text{Perms} = (0,1] \\ \text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} (\text{Values} \times \text{Perms}) & \text{Vars} = \{x,y,\ldots\} \\ \text{States} = (\text{Stacks} \times \text{Heaps}) \cup \{\top\} & D = \mathcal{P}(\text{States}) \end{array}$$

Figure 7. The domain D of the concrete semantics

for this language. As was previously noted, in our fixed-point characterization of resource invariants for heap-manipulating programs, we assume that the borders between parts of the heap owned by different threads or protected by different locks are stable, i.e., pointed to by fixed stack variables. We thus have to account for the fact that local state of several threads and resource invariants of several locks may reference the same variable. Supposing that there are k such locks and threads, we handle this case by giving to each thread and lock a fractional permission $1/k$ for this variable [2]. The permission shows "how much" of this variable is owned by the thread or protected by the lock. The idea is that a thread having a permission less than 1 for a variable can read it; a thread can write to a variable only if the permission associated with it in its local state is equal to 1, i.e., only if it gathers the permissions from all the other locks that own this variable by acquiring them. Thus, as our concrete domain we chose an extension of the example of a separation domain presented in Section 3.2 with fractional permissions for variables.

The domain D is defined in Figure 7. As before, we identify all the sets of states from States containing $\top$ and denote such elements of the domain simply with $\top$. The order in the domain is subset inclusion with $\top$ being the topmost element, the join operation is set union, and the bottom element is the empty set.

We proceed to define an operation of separate combination $*$ on the domain. Informally, $*$ adds up permissions for variables and computes the disjoint combination of heaps. It corresponds to the model of the separating conjunction from separation logic in the case when variables are treated as resources with permissions.

For $t \in \text{Stacks}$ let functions Val_t and Perm_t be selectors for values, respectively, permissions, of variables on the stack: for all $x \in \text{Vars}$, $(\mathsf{Val}_t(x), \mathsf{Perm}_t(x)) = t(x)$ and, additionally, $\mathsf{Perm}_t(x) = 0$ if $t(x)\uparrow$.

We define the combination $t_1 * t_2$ of stacks $t_1, t_2 \in \text{Stacks}$ as follows: if $\forall x \in \text{Vars}.\ \mathsf{Perm}_{t_1}(x) + \mathsf{Perm}_{t_2}(x) \leq 1 \wedge (\mathsf{Val}_{t_1}(x)\downarrow \wedge \mathsf{Val}_{t_2}(x)\downarrow \Rightarrow \mathsf{Val}_{t_1}(x) = \mathsf{Val}_{t_2}(x))$, then

$$(t_1 * t_2)(x) = \begin{cases} (\mathsf{Val}_{t_1}(x), \mathsf{Perm}_{t_1}(x) + \mathsf{Perm}_{t_2}(x)), & \text{if } \mathsf{Val}_{t_1}(x)\downarrow; \\ (\mathsf{Val}_{t_2}(x), \mathsf{Perm}_{t_1}(x) + \mathsf{Perm}_{t_2}(x)), & \text{if } \mathsf{Val}_{t_2}(x)\downarrow; \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

and $t_1 * t_2$ is undefined otherwise.

We define the combination $h_1 * h_2$ of heaps $h_1, h_2 \in \text{Heaps}$ in the following way:

$$h_1 * h_2 = \begin{cases} h_1 \uplus h_2, & \text{if } \mathsf{dom}(h_1) \cap \mathsf{dom}(h_2) = \emptyset; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Finally, for $t_1, t_2 \in \text{Stacks}$ and $h_1, h_2 \in \text{Heaps}$ we let $(t_1, h_1) * (t_2, h_2) = (t_1 * t_2, h_1 * h_2)$, for $s \in D$ we let $\top * s = \top * s = \top$, and we lift $*$ to D pointwise.

The unit element e of $*$ is the singleton set containing a pair of an empty heap and an empty stack, both represented by everywhere undefined functions.

Transfer functions. We define the concrete transfer functions $f_C : D \to D$ for primitive commands using the transition relation $\rightsquigarrow$ shown in Figure 8. We let $f_C(t,h) = \{s \mid C, t, h \rightsquigarrow s\}$, $f_C(\top) = \top$, and lift f_C to D pointwise. It is not difficult to show that for each primitive command C from Figure 6 the transfer function f_C is monotone and local.

$$\begin{array}{ll} x = y, t, h \rightsquigarrow t[x : (\mathsf{Val}_t(y), 1)], h & \text{if } \mathsf{Perm}_t(x) = 1, t(y)\downarrow \\ x = \mathsf{null}, t, h \rightsquigarrow t[x : (0,1)], h & \text{if } \mathsf{Perm}_t(x) = 1 \\ x = y{\rightarrow}\mathit{next}, t, h \rightsquigarrow t[x : h(\mathsf{Val}_t(y))], h & \text{if } \mathsf{Perm}_t(x) = 1, \\ & t(y)\downarrow, h(\mathsf{Val}_t(y))\downarrow \\ x{\rightarrow}\mathit{next} = y, t, h \rightsquigarrow t, h[\mathsf{Val}_t(x) : \mathsf{Val}_t(y)] & \text{if } t(x)\downarrow, t(y)\downarrow, \\ & h(\mathsf{Val}_t(x))\downarrow \\ x{\rightarrow}\mathit{next} = \mathsf{null}, t, h \rightsquigarrow t, h[\mathsf{Val}_t(x) : 0] & \text{if } t(x)\downarrow, h(\mathsf{Val}_t(x))\downarrow \\ x = \textbf{new}, t, h \rightsquigarrow t[x : a], h[a : e] & \text{if } \mathsf{Perm}_t(x) = 1, \\ & a \in \text{Locs} \setminus \mathsf{dom}(h), \\ & e \in \text{Values} \\ \textbf{delete}\ x, t, h[\mathsf{Val}_t(x) : e] \rightsquigarrow t, h & \text{if } t(x)\downarrow, e \in \text{Values} \\ \textbf{assume}(G), t, h \rightsquigarrow t, h & \text{if } [\![G]\!]t\downarrow, [\![G]\!]t = \mathsf{true} \\ \textbf{assume}(G), t, h \not\rightsquigarrow & \text{if } [\![G]\!]t\downarrow, [\![G]\!]t = \mathsf{false} \\ C, t, h \rightsquigarrow \top & \text{otherwise} \end{array}$$

Figure 8. Transition relation for primitive commands. $[\![G]\!]t \in \{\mathsf{true}, \mathsf{false}\}$ is the valuation of the guard G over the stack t (which is undefined if t is undefined for a variable used in G). Here $\not\rightsquigarrow$ denotes that the command does not fault, but gets stuck.

4.2 Fixed-point characterization

In the setup of Section 3.3 we let both the abstract separation domain $D^\sharp$ and the concrete D be equal to the domain defined in Section 4.1 and the abstract transfer functions be equal to the concrete ones. γ is the identity function. We also let $\mathsf{pre}^\sharp = \mathsf{pre}$. All the conditions listed at the end of Section 3.2 are satisfied.

We would like to characterize resource invariants as least fixed points of a functional from the class defined in Figure 5. To do this we have to provide functions Local_i and Frame_i ($i = 1..n$) for each lock, Pre_i ($i = 1..m$) for each thread, and Init_i ($i = 1..n$) for each lock.

To this end we assume that for each lock ℓ_i we are given two sets of program variables, which specify the borders of the part of the heap protected by the lock—the set of *entry points* $\mathsf{Entry}(\ell_i)$ and the set of *exit points* $\mathsf{Exit}(\ell_i)$. The part of the heap protected by the lock is defined as everything that is reachable from entry points up to exit points. Therefore, exit points are pointers that lead to parts of the heap owned by others. In addition, for each thread P_i we assume given the set $\mathsf{Owns}(P_i)$ of program variables that it owns. These are variables that the thread is allowed to access even without holding any locks. We define $\mathsf{Owns}(\ell_i) = \mathsf{Entry}(\ell_i) \cup \mathsf{Exit}(\ell_i)$.

First, we define how Local_i and Frame_i partition the stack. To this end we define an auxiliary function that describes the ownership partitioning of the stack. For a thread or lock P, Perm^0_P defines via permissions the part of the stack owned by P: for each $x \in \text{Vars}$

$$\mathsf{Perm}^0_P(x) = \begin{cases} 1/|\{P' \mid x \in \mathsf{Owns}(P')\}|, & \text{if } x \in \mathsf{Owns}(P); \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

For a stack $t \in \text{Stacks}$ if $\forall x \in \text{Vars}.\ t(x)\downarrow \Rightarrow \mathsf{Perm}^0_{\ell_i}(x) \leq \mathsf{Perm}_t(x)$, then let

$$\mathsf{Local}_i(t) = \lambda x.(\mathsf{Val}_t(x), \mathsf{Perm}_t(x) - \mathsf{Perm}^0_{\ell_i}(x)),$$
$$\mathsf{Frame}_i(t) = \lambda x.(\mathsf{Val}_t(x), \mathsf{Perm}^0_{\ell_i}(x)),$$

otherwise let $\mathsf{Local}_i(t) = \mathsf{Frame}_i(t) = \top$.

We now define how Local_i and Frame_i partition the heap. Frame_i takes the part of the heap reachable from entry points up to exit points. The intuition is that modifying pointers to a heap cell such that it moves into this part of the heap means that it becomes protected by the corresponding lock and a part of its resource invariant (recall that in the analysis Frame_i computes the part of

the heap that goes into the resource invariant). The rest of the heap becomes the result of Local_i.

For $t \in \text{Stacks}$, $h \in \text{Heaps}$, and a lock ℓ_i let $d(\ell_i, t, h)$ be the smallest set d such that $(\mathsf{Val}_t(\mathsf{Entry}(\ell_i)) \cup (h(d) \setminus \mathsf{Val}_t(\mathsf{Exit}(\ell_i)))) \cap \text{Locs} \subseteq d$. $d(\ell_i, t, h)$ defines the part of the heap reachable from the entry points of the lock ℓ_i up to exit points associated with this lock. We now let for $t \in \text{Stacks}$ and $h \in \text{Heaps}$

$$\mathsf{Local}_i(t, h) = (\mathsf{Local}_i(t), h|_{\mathsf{dom}(h) \setminus d(\ell_i, t, h)}),$$
$$\mathsf{Frame}_i(t, h) = (\mathsf{Frame}_i(t), h|_{d(\ell_i, t, h)}).$$

Here we assume that $(\top, h) = \top$. We let $\mathsf{Local}_i(\top) = \mathsf{Frame}_i(\top) = \top$ and lift Local_i and Frame_i to D pointwise.

We proceed to determine the initial splitting of the heap among threads and locks. This is done in the same spirit as splitting the heap at **release** commands. For each thread P_i we define $\mathsf{Entry}(P_i) = \mathsf{Owns}(P_i)$, $\mathsf{Exit}(P_i) = \{\mathsf{Owns}(P_k) \mid 1 \le i \le m \wedge i \neq k\} \cup \{\mathsf{Entry}(\ell_i) \mid 1 \le i \le n\}$. That is, for a thread P_i we consider the entry points of its initial states to be the variables owned by the thread and exit points to be the variables owned by other threads and entry points of locks. For a thread P_i let $d(P_i, t, h)$ be defined as before, but with new Entry and Exit. For a stack t and a heap h we let

$$\mathsf{Pre}_i(t, h) = (\lambda x.(\mathsf{Val}_t(x), \mathsf{Perm}^0_{P_i}(x)), h|_{d(P_i, t, h)}),$$
$$\mathsf{Init}_i(t, h) = (\lambda x.(\mathsf{Val}_t(x), \mathsf{Perm}^0_{\ell_i}(x)), h|_{d(\ell_i, t, h)}).$$

For $s \in D$ we define

$$\mathsf{Pre}_i(s) = \{\mathsf{Pre}_i(t, h) \mid (t, h) \in s\}, \quad \mathsf{Pre}_i(\top) = \top,$$
$$\mathsf{Init}_i(s) = \{\mathsf{Init}_i(t, h) \mid (t, h) \in s\}, \quad \mathsf{Init}_i(\top) = \top,$$

if for each $(t, h) \in s$

$$(\uplus_{i=1}^{m} \mathsf{dom}(\mathsf{Pre}_i(t, h))) \uplus (\uplus_{i=1}^{n} \mathsf{dom}(\mathsf{Init}_i(t, h))) = \mathsf{dom}(h),$$

where $\mathsf{dom}(t, h) = \mathsf{dom}(h)$ and $\mathsf{Pre}_i(s) = \mathsf{Init}_i(s) = \top$ otherwise. The latter case corresponds to the situation when we are unable to split the initial states using the definitions of Entry and Exit above.

Since the transfer functions f_C and functions Local_i, Frame_i Pre_i, and Init_i are monotone, by Tarski's fixed point theorem the functional $F^\sharp$ defined in Figure 5 given the instantiations above always has least fixed point (Q, I). It is not difficult to show that Local_i and Frame_i defined above satisfy (6) and Pre_i and Init_i defined above satisfy (7). Hence, from Theorem 1 it follows that (Q, I) over-approximates the concrete semantics (as defined in Figure 3) and provides a valid fixed-point characterization of resource invariants for a class of heap-manipulating programs.

Example. The program in Figure 9 is adapted from [19]. Two threads—a producer and a consumer—use fine-grained synchronization to access an unbounded buffer represented by a singly-linked list. Variables *first* and *last* point to the first, respectively, last element of the list. Variable x is a local variable of the consumer, y of the producer. The *last* element is a dummy node. The producer adds a portion to the buffer by placing that portion in the *data* field of *last*, then allocates a new cell, and links that cell into the list making it the *last* (and dummy). Removing a portion results in a value being read from the *data* field of the *first* node, disposing this node, and moving *first* along the list by one. When the list is of length one *last* and *first* are equal, and this corresponds to an empty buffer; it is in this case that synchronization is necessary. In this example $\mathsf{Entry}(\ell) = \{\mathit{first}\}$, $\mathsf{Exit}(\ell) = \{\mathit{last}\}$, $\mathsf{Owns}(\mathit{producer}) = \{\mathit{last}, y\}$, and $\mathsf{Owns}(\mathit{consumer}) = \{\mathit{first}, x\}$. Figure 10 defines a fixed point of the functional introduced above. We use separation logic formulae to denote elements of the domain D (in fact, the formulae in Figure 10 represent an outline of a proof in concurrent separation logic). The only deviation from the standard syntax and semantics [23] is that we treat variables as resources [21] (as required by the semantics in Section 4.1). The syntax of formulae is the same as in Figure 2 except for the following:

- Each formula is prefixed with an expression of the form $p_1 x_1, \ldots, p_k x_k$ describing the stack, where p_i is a number in $(0, 1]$ representing the permission for the variable x_i and $x_1, \ldots, x_k$ is the list of free variables in the formula.
- We use $E_1 \mapsto \{\mathit{next}: E_2\}$ instead of $E \mapsto \{\mathit{prev}: E, \mathit{next}: E\}$.
- We use a predicate $\mathsf{ls}(E_1, E_2)$ for singly-linked lists rather than a predicate for doubly-linked lists.

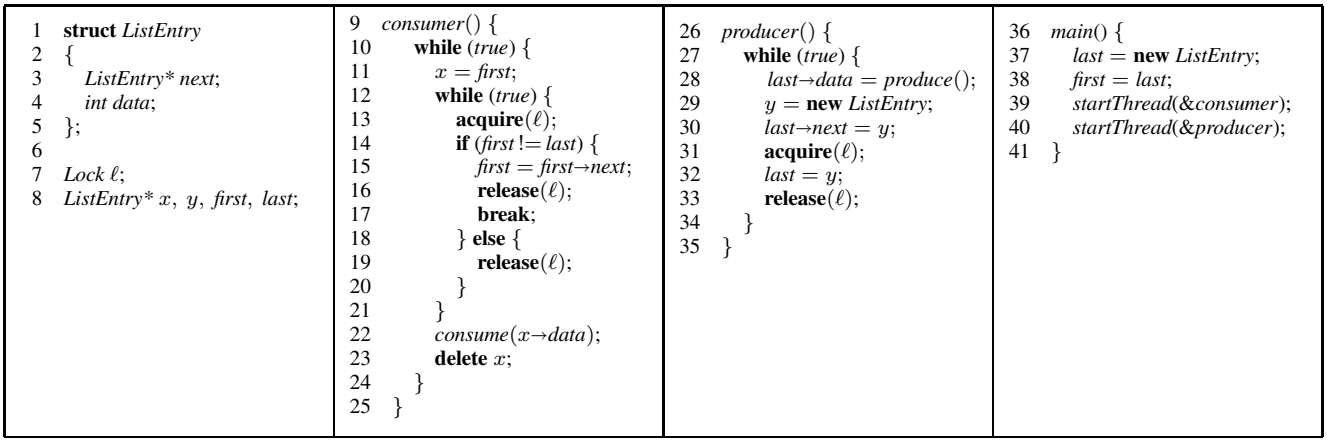
We contract $1x$ to simply x, $E_1 \mapsto \{\mathit{next}: E_2\}$ to $E_1 \mapsto E_2$, and use $x \mapsto \_$ instead of $x \mapsto x'$ when the value of x' is irrelevant. $\mathsf{ls}(E_1, E_2)$ is the least predicate such that $\mathsf{ls}(E_1, E_2) \Leftrightarrow \exists x.(E_1 = E_2 \wedge \mathsf{emp}) \vee (E_1 \neq E_2 \wedge E_1 \mapsto x * \mathsf{ls}(x, E_2))$ and denotes all of the states in which the heap has the shape of an acyclic (possibly empty) singly-linked list, with the head node E_1 and the value of the *next* field of the last node E_2. Note that E_2 must be a dangling pointer in a heap defined by $\mathsf{ls}(E_1, E_2)$. The adjustments to the semantics of formulae for handling variables as resources are as in [21]. The formula for the line v corresponds to the value of $Q(v, L)$, where L is the set of locks held at v by the thread to which v belongs (in this example for each CFG node v the set L can be determined syntactically). The resource invariant associated with the lock ℓ is $\frac{1}{2}\mathit{first}, \frac{1}{2}\mathit{last} \Vdash \mathsf{ls}(\mathit{first}, \mathit{last})$. We omitted formulae for some program lines to conserve space. □

5. Instantiating the framework

By instantiating the framework for thread-modular analyses presented in Section 3.3 with the concrete domain defined in Section 4.1, and abstract domain and transfer functions from different sequential shape analyses we can get different thread-modular shape analyses. In this section we present an instantiation of the framework on which our implementation is based and outline two other instantiations. Their design follows the fixed-point characterization of resource invariants presented in Section 4.2, i.e., the analyses split the heap on releasing a lock by computing (an approximation of) reachability between entry and exit points. We first note an important property of thread-modular shape analyses obtained by instantiating the framework with the concrete domain defined in Section 4.1—they detect data races, i.e., having a data race in the program results in the analysis signaling a possible bug.

5.1 Data race freedom

Suppose we are given a thread-modular shape analysis obtained from an instantiation of the framework of Section 3.3 in which the concrete separation domain D is equal to the domain defined in Section 4.1. Thus, we assume given an abstract separation domain $D^\sharp$, a concretization function γ, abstract transfer functions $f^\sharp_C$, and functions $\mathsf{Local}^\sharp_i$, $\mathsf{Frame}^\sharp_i$, $\mathsf{Pre}^\sharp_i$, and $\mathsf{Init}^\sharp_i$. We fix a program in the programming language introduced in Section 4.1 and let q be least fixed point of the functional F (Figure 3) that defines the concrete semantics of the program. We prove that the success of the thread-modular shape analysis on the program implies that the program has no data races (both on stack variables and on heap cells). The analysis succeeds if the fixed point $(Q^\sharp, I^\sharp)$ of the functional $F^\sharp$ (Figure 5) that defines its result is such that for every node v and every set of locks L it is the case that $Q^\sharp(v, L) \sqsubset \top$ and for all locks ℓ_i it is the case that $I^\sharp_i \sqsubset \top$. We denote this with $(Q^\sharp, I^\sharp) \sqsubset \top$. In other words, a possible error found by the analysis is denoted by the topmost element in the abstract domain. This requires us to put an additional constraint on the abstract



```
1  struct ListEntry
2  {
3    ListEntry* next;
4    int data;
5  };
6
7  Lock ℓ;
8  ListEntry* x, y, first, last;

9  consumer() {
10   while (true) {
11     x = first;
12     while (true) {
13       acquire(ℓ);
14       if (first != last) {
15         first = first→next;
16         release(ℓ);
17         break;
18       } else {
19         release(ℓ);
20       }
21     }
22     consume(x→data);
23     delete x;
24   }
25 }

26 producer() {
27   while (true) {
28     last→data = produce();
29     y = new ListEntry;
30     last→next = y;
31     acquire(ℓ);
32     last = y;
33     release(ℓ);
34   }
35 }

36 main() {
37   last = new ListEntry;
38   first = last;
39   startThread(&consumer);
40   startThread(&producer);
41 }
```

Figure 9. Example program

Line	Local state
10	$x, \frac{1}{2}first \Vdash \mathsf{emp}$
11	$x, \frac{1}{2}first \Vdash \mathsf{emp}$
13	$x, \frac{1}{2}first \Vdash x = first \wedge \mathsf{emp}$
14	$x, first, \frac{1}{2}last \Vdash x = first \wedge \mathsf{ls}(first, last)$
16	$x, first, \frac{1}{2}last \Vdash x{\mapsto}first * \mathsf{ls}(first, last)$
17	$x, \frac{1}{2}first \Vdash x{\mapsto}first$
19	$x, first, \frac{1}{2}last \Vdash first = last \wedge x = first \wedge \mathsf{ls}(first, last)$
20	$x, \frac{1}{2}first \Vdash x = first \wedge \mathsf{emp}$
21	$x, \frac{1}{2}first \Vdash x = first \wedge \mathsf{emp}$
23	$x, \frac{1}{2}first \Vdash x{\mapsto}first$
24	$x, \frac{1}{2}first \Vdash \mathsf{emp}$
28	$y, \frac{1}{2}last \Vdash last{\mapsto}\_$
30	$y, \frac{1}{2}last \Vdash last{\mapsto}\_ * y{\mapsto}\_$
31	$y, \frac{1}{2}last \Vdash last{\mapsto}y * y{\mapsto}\_$
32	$y, last, \frac{1}{2}first \Vdash \mathsf{ls}(first, last) * last{\mapsto}y * y{\mapsto}\_$
33	$y, last, \frac{1}{2}first \Vdash y = last \wedge (\mathsf{ls}(first, last) * last{\mapsto}\_)$
34	$y, \frac{1}{2}last \Vdash last{\mapsto}\_$
37	$x, y, first, last \Vdash \mathsf{emp}$
39	$x, y, first, last \Vdash first = last \wedge last{\mapsto}\_$

Figure 10. The fixed point defined by the fixed-point characterization for the program in Figure 9. Results for some lines of the program are elided.

domain: we require that

$$\forall s \in D^\sharp.\ \gamma(s) = \top \Rightarrow s = \top. \tag{8}$$

Let $\mathsf{accesses}(C, t, h)$, respectively, $\mathsf{writes}(C, t, h)$ be the set of variables and locations that the primitive sequential command C may access (i.e., read, write or dispose), respectively, write to or dispose, when run from the state (t, h).

DEFINITION 3 (Interfering commands). *Primitive sequential commands C_1 and C_2 interfere with each other when executed from the state (t, h), denoted with $C_1 \bowtie_{(t,h)} C_2$, if $\mathsf{accesses}(C_1, t, h) \cap \mathsf{writes}(C_2, t, h) \neq \emptyset$ or $\mathsf{writes}(C_1, t, h) \cap \mathsf{accesses}(C_2, t, h) \neq \emptyset$.*

Given this formulation of interference, the usual notion of data races is formulated as follows.

DEFINITION 4 (Data race). *The program has a data race if for some location $(v_1, \ldots, v_m)$, admissible lockset $(L_1, \ldots, L_m)$, and state $(t, h) \in q(v_1, \ldots, v_m, L_1, \ldots, L_m) \sqsubset \top$ there exist CFG edges $(v_1, C_1, v_1') \in E_i$ and $(v_2, C_1, v_2') \in E_j$ $(i \neq j)$ labeled with sequential commands C_1 and C_2 such that $C_1, t, h \not\rightsquigarrow \top$, $C_2, t, h \not\rightsquigarrow \top$ and $C_1 \bowtie_{(t,h)} C_2$.*

THEOREM 2 (Data race freedom). *Suppose that $(Q^\sharp, I^\sharp)$ is a fixed point of the functional $F^\sharp$ obtained from an instantiation of the framework from Section 3.3 with the concrete domain from Section 4.1 and a concretization function γ satisfying (8). If $(Q^\sharp, I^\sharp) \sqsubset \top$, then the program has no data races.*

The proof appears in Appendix B.

5.2 Thread-modular shape analysis with separated heap abstractions

We define a thread-modular analysis based on a sequential shape analysis in which abstract states are represented by separation logic formulae. The underlying sequential analysis is specialized for handling doubly-linked lists and is similar to the one of [9].

The abstract domain $D^\sharp$ is the domain of sets of separation logic formulae from the subset defined in Section 2 (Figure 2) and extended to handle variables as resources as described at the end of Section 4.2. A special element $\top \in D^\sharp$ denotes an error. The order on the domain is subset inclusion with $\top$ being the topmost element. The concrete domain is the domain from Section 4.1 modified in the obvious way to account for multiple field selectors in structures. The concretization function is defined following the semantics of separation logic formulae. The operation of separate combination $\sharp$ on the abstract domain is just separating conjunction $*$ lifted to the sets of formulae. Transfer functions are similar to those presented in [9, 12], but adapted for handling doubly-linked lists rather than singly-linked lists. Functions $\mathsf{Local}_i^\sharp$, $\mathsf{Frame}_i^\sharp$, $\mathsf{Pre}_i^\sharp$, and $\mathsf{Init}_i^\sharp$ mirror their concrete counterparts defined in Section 4.2. Their implementation is similar to the implementation of the operations used in interprocedural shape analysis with an abstract representation based on separation logic [12] to split the abstract heap at a procedure call (functions local and frame in [12]). The difference with respect to the concrete operations is that instead of computing reachability precisely in the concrete, we compute its approximation—reachability in the formula as defined in [12]. The analysis performs the sequential shape analysis of the initialization code to obtain an abstract precondition $\mathsf{pre}^\sharp$ of the program. The reader may now revisit the example in Section 2 to get the idea

of how the analysis works. Note that to simplify presentation we elided the discussion of treating variables as resources in the analysis while presenting the example.

We note that in situations when all of the heap cells in the data structure protected by a lock are reachable from some set of program variables, the sets of exit points are empty and the set of program variables protected by a lock forms a reasonable guess for the set of entry points associated with the lock. The set of entry points can then be inferred using, e.g., tools (both static and dynamic) for analyzing correlations between locks and variables that determine the set of locks that are held consistently each time a variable is accessed [22, 25, 5].

5.3 Other instantiations

Distefano et al. [9] present a shape analysis for singly-linked lists in which abstract states are represented by separation logic formulae. The construction of a thread-modular shape analysis that uses [9] as the underlying sequential analysis is done in the same way as described in Section 5.2. Unlike the domain presented in Section 5.2, the domain from [9] is finite provided the number of program variables is bounded. Hence, the corresponding thread-modular analysis is always guaranteed to terminate, which is not the case for the analysis presented in Section 5.2.

Lev-Ami et al. [16] present a shape analysis that handles a wider range of data structures including singly- and doubly-linked lists and binary trees. Abstract states in the analysis are represented by shape graphs. To define functions $\mathsf{Local}_i^\sharp$, $\mathsf{Frame}_i^\sharp$, $\mathsf{Pre}_i^\sharp$, and $\mathsf{Init}_i^\sharp$ for this abstract domain we have to be able to split shape graphs creating dangling pointers across splittings. The abstract representation of [16] does not allow dangling pointers, but can be extended [15] so that it is suitable for us in a restricted case in which the set of exit points for each lock is empty and each stack variable is owned by only one process or resource (i.e., sets Entry and Owns are pairwise disjoint). More precisely, as the concrete separation domain we again take the domain from Section 4.1. The abstract domain is represented by a topped powerset of shape graphs extended to allow special *unusable pointers*. These are pointers such that no information about them is preserved by the analysis and dereferencing them results in an error. Adding them preserves the finiteness of the domain. The operation of separate combination is the union of shape graphs. Functions $\mathsf{Local}_i^\sharp$, $\mathsf{Frame}_i^\sharp$, $\mathsf{Pre}_i^\sharp$, and $\mathsf{Init}_i^\sharp$ are defined following their concrete counterparts in Section 4.2 using reachability in shape graphs. Since we assume empty sets of exit points, there may only be two kinds of dangling pointers: dangling pointers resulting from the **delete** command and pointers from the part of the heap computed by $\mathsf{Frame}_i^\sharp$ to the part of the heap computed by $\mathsf{Local}_i^\sharp$. Both of these kinds of pointers can be modeled in the abstract representation by unusable pointers.

6. Implementation and experimental results

We have implemented the thread-modular shape analysis described in Section 5.2 in a prototype tool and applied it to multithreaded heap-manipulating code from Windows device drivers. The results from our experiments are presented in Figure 11. Tests were performed on a 3.4GHz Pentium 4 PC. In all of our experiments the maximum memory usage by the tool was 22MB. The sizes of actual C code (without comments, irrelevant definitions, etc.) that was analyzed for examples from Figure 11 ranged from 50 to 300 LOC.

Each program we attempted to verify consisted of 2-6 threads representing concurrently executing dispatch routines of device drivers that performed different operations on doubly-linked lists. The precondition of each thread was just the empty heap. Hence, according to the note at the end of Section 3.3, the results of our analysis are also valid for an unbounded number of copies of threads that are present in the code. In all cases we let the sets of exit points for all locks be empty. The tool found bugs in three programs and proved that the rest of programs are memory-safe, do not leak memory and are data race free. In each case the analysis converged within a few iterations. We have not encountered any false bugs in our experiments. The bugs found by the analysis were due to accesses to data structures not protected by locks and would manifest themselves as dereferencing a dangling pointer or a memory leak.

Threads	3	6	9	12	15	18
Time (sec)	11.4	27.7	50.3	79.9	118.7	170.7

Figure 12. Results of testing the tool on programs with varying number of threads

To speed up convergence to a fixed point in our implementation we use a non-standard widening operator [8, 4] that eliminates redundant formulae from sets of formulae representing abstract states using a prover for entailment between separation logic formulae similar to the one described in [1]. The proof of Theorem 1 can be adjusted to ensure the soundness of the analysis using the widening operator. Due to the use of the non-standard widening, in most cases the final number of states per program point was 1 or 2.

To assess the scalability of our shape analysis we took program 2 and duplicated the code of threads in it knowing that the analyzer will not realize that the threads are duplicates. The results are presented in Figure 12. As can be seen from the figure, we do not observe the analysis being exponential in the number of threads.

7. Related work

The shape analysis for concurrent programs presented in [26] handles unbounded numbers of locks (in the heap) and threads but relies on abstracting program interleaving and thus does not scale well. A number of analyses have been developed to detect races in multithreaded programs, both automatic (e.g., [25, 5, 18, 22, 17]) and requiring user annotations (e.g., [10, 3, 14]). To the best of our knowledge all of the automatic tools are either overly imprecise or unsound in the presence of deep heap update. The analyses that require annotations, which are usually based on type systems, preclude ownership transfer; besides, the annotations required by them are often too heavyweight. In contrast, our analysis handles ownership transfer and requires lightweight annotations that can be inferred by existing automatic tools. Some of the techniques for race detection (including [22, 25, 5]) provide information about which locks protect which variables. Such techniques are complimentary to ours—they can be used to discover entry points for resource invariants needed by our analysis.

Our method for constructing thread-modular shape analyses is inspired by concurrent separation logic [19], which adapted the idea of resource invariants to heap-manipulating programs. Here we use this idea in the context of program analysis. However, it is important to note a difference between the approach we are taking in this paper and the approach that is taken in concurrent separation logic. In concurrent separation logic resource invariants have to be precise—informally, they have to unambiguously pick out an area of heap; see [19] for a formal definition. The reason is that having imprecise resource invariants leads to the possibility of choosing different splittings of the heap at **release** commands in a proof, which makes the conjunction rule of Hoare logic unsound. Here the determinism of heap splittings at **release** commands is enforced by using deterministic functions Local_i and Frame_i. Hence, in our analysis we can compute resource invariants without worrying about their precision and still keep the analysis sound.

Test	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time (sec)	2.2	11.7	3.4	4.2	3.5	6.6	8.6	13.3	6.9	6.6	27.2	28.6	9.7	5.3	3.4	5.3
Bug found	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes

Figure 11. Results of the application of the tool to multithreaded code from Windows device drivers

Previous work [6] presented a fixed-point characterization of resource invariants for integer programs with semaphores in the case when programs have no shared variables. This paper is the first, to the best of our knowledge, to present a fixed-point characterization of a class of resource invariants for heap-manipulating programs.

8. Conclusion

We have described a new analysis designed to eliminate the consideration of interleavings for programs with deep heap update while preserving soundness and precision. Our analysis is able to establish that the program is memory-safe (i.e., it does not dereference heap cells that are not allocated), does not leak memory and does not have data races (including races on heap cells). The analysis handles low-level language features including non-lexically scoped and nested locking, and memory disposal. Our solution works particularly well in situations in which all of the heap cells in the data structure protected by a lock are reachable from some set of program variables as it is the case, e.g., in systems code.

Acknowledgements. We would like to thank Cristiano Calcagno, Dino Distefano, Peter O'Hearn, Tal Lev-Ami, Stephen Magill, Roman Manevich, Matthew Parkinson, Andreas Podelski, Zvonimir Rakamaric, Ganesan Ramalingam, John Reynolds, Noam Rinetzky, Viktor Vafeiadis, Hongseok Yang, Jian Zhang, and the anonymous reviewers for comments and discussions that helped to improve the paper.

References

[1] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS'05: Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[2] R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05: Principles of Programming Languages*, pages 259–270. ACM Press, 2005.

[3] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02: Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM Press, 2002.

[4] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS'06: Static Analysis Simposium*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[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 *PLDI'02: Programming Languages Design and Implementation*, pages 258–269. ACM Press, 2002.

[6] E. Clarke. Synthesis of resource invariants for concurrent programs. *ACM Trans. Program. Lang. Syst.*, 2(3):338–358, 1980.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[8] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[9] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06: Tools and Algorithms for Analysis and Construction of Systems*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[10] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI'00: Programming Languages Design and Implementation*, pages 219–232. ACM Press, 2000.

[11] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN'03: Workshop on Model Checking Software*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.

[12] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS'06: Static Analysis Symposium*, volume 4134 of *LNCS*, pages 240–260. Springer, 2006.

[13] A. Gotsman, N. Rinetzky, J. Berdine, B. Cook, D. Distefano, P. W. O'Hearn, M. Sagiv, and H. Yang. Abstract interpretation with state separation. In preparation, 2007.

[14] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI'03: Types in Languages Design and Implementation*, pages 13–25. ACM Press, 2003.

[15] T. Lev-Ami. Personal communication. 2006.

[16] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV'06: Computer Aided Verification*, volume 4144 of *LNCS*, pages 547–561. Springer, 2006.

[17] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL'07: Principles of Programming Languages*, pages 327–338. ACM Press, 2007.

[18] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI'06: Programming Languages Design and Implementation*, pages 308–319. ACM Press, 2006.

[19] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04: International Conference on Concurrency Theory*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.

[20] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–284, 1976.

[21] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS'06: Logic in Computer Science*, pages 137–146. IEEE Press, 2006.

[22] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI'06: Programming Languages Design and Implementation*, pages 320–331. ACM Press, 2006.

[23] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Logic in Computer Science*, pages 55–74. IEEE Press, 2002.

[24] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS'05: Static Analysis Symposium*, volume 3672 of *LNCS*, pages 284–302. Springer, 2005.

[25] S. Savage, M. Burrows, G. Nelson, P. Soblvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Comp. Syst.*, 15(4):371–411, 1997.

[26] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL'01: Principles of Programming Languages*, pages 27–40. ACM Press, 2001.

[27] H. Yang and P. W. O'Hearn. A semantic basis for local reasoning. In *FOSSACS'02: Foundations of Software Science and Computation Structures*, volume 2303 of *LNCS*, pages 402–416. Springer, 2002.

A. Proof of Theorem 1 (Soundness)

Let $\tilde{q} \in \widehat{D}$ be defined as follows: for each location $(v_1, \ldots, v_m)$ and admissible lockset $(L_1, \ldots, L_m)$

$$\tilde{q}(v_1, \ldots, v_m, L_1, \ldots, L_n) = \gamma\left(\left(\bigsharp_{i=1}^{m} Q(v_i, L_i)\right) \sharp \left(\bigsharp_{\ell_i \notin L} I_i\right)\right)$$

(where $L = L_1 \cup \ldots \cup L_m$) and for each location $(v_1, \ldots, v_m)$ and inadmissible lockset $(L_1, \ldots, L_m)$, $\tilde{q}(v_1, \ldots, v_m, L_1, \ldots, L_n) = \bot$. We show that $F(\tilde{q}) \sqsubseteq \tilde{q}$. By Park induction principle this implies that $q = \mathsf{lfp}(F) \sqsubseteq \tilde{q}$, which is required.

First of all, from (5) and (7) it follows that

$$(F(\tilde{q}))(\mathsf{start}_1, \ldots, \mathsf{start}_m, \emptyset, \ldots, \emptyset) \sqsubseteq \tilde{q}(\mathsf{start}_1, \ldots, \mathsf{start}_m, \emptyset, \ldots, \emptyset).$$

According to the definition of the functional F (Figure 3), it is now sufficient to show that for all locations $(v_1, \ldots, v_j, \ldots, v_m)$, admissible locksets $(L_1, \ldots, L_m)$, and edges $(v_j^0, C, v_j) \in E$ it is the case that

$$g_C^j(\tilde{q}(v_1, \ldots, v_j^0, \ldots, v_m), L_1, \ldots, L_m) \sqsubseteq \tilde{q}(v_1, \ldots, v_j, \ldots, v_m, L_1, \ldots, L_m). \quad (9)$$

There are three cases corresponding to the type of the command C.

Case 1. C is a sequential command. We have to show that

$$f_C(\tilde{q}(v_1, \ldots, v_j^0, \ldots, v_m, L_1, \ldots, L_m)) \sqsubseteq \tilde{q}(v_1, \ldots, v_j, \ldots, v_m, L_1, \ldots, L_m). \quad (10)$$

Let $L = L_1 \cup \ldots \cup L_m$, $s_1 = Q(v_j^0, L_j)$, $s_2 = Q(v_j, L_j)$, and

$$s = \left(\bigsharp_{\substack{1 \le i \le m, \\ i \ne j}} Q(v_i, L_i)\right) \sharp \left(\bigsharp_{\ell_i \notin L} I_i\right). \quad (11)$$

Then (10) is equivalent to $f_C(\gamma(s \sharp s_1)) \sqsubseteq \gamma(s \sharp s_2)$. Since (Q, I) is a fixed point of the functional $F^\sharp$, by the definition of $F^\sharp$ (Figure 5) we get

$$f_C^\sharp(s_1) \sqsubseteq s_2. \quad (12)$$

Then, since $*$ and γ are monotone,

$$\begin{aligned} f_C(\gamma(s \sharp s_1)) &= f_C(\gamma(s) * \gamma(s_1)) && \text{by (3)} \\ &\sqsubseteq \gamma(s) * f_C(\gamma(s_1)) && \text{by (2)} \\ &\sqsubseteq \gamma(s) * \gamma(f_C^\sharp(s_1)) && \text{by (4)} \\ &= \gamma(s \sharp f_C^\sharp(s_1)) && \text{by (3)} \\ &\sqsubseteq \gamma(s \sharp s_2) && \text{by (12)} \end{aligned}$$

Case 2. C is **acquire**(ℓ_k). We can assume that $\ell_k \in L_j$, otherwise the left-hand side of (9) is $\bot$. Thus, we have to show that

$$\tilde{q}(v_1, \ldots, v_j^0, \ldots, v_m, L_1, \ldots, L_j \setminus \{\ell_k\}, \ldots, L_m) \sqsubseteq \tilde{q}(v_1, \ldots, v_j, \ldots, v_m, L_1, \ldots, L_j, \ldots, L_m). \quad (13)$$

Let s be defined by (11) with $L = L_1 \cup \ldots \cup L_m$ and let $s_1 = Q(v_j^0, L_j \setminus \{\ell_k\})$ and $s_2 = Q(v_j, L_j)$. Since the lockset $(L_1, \ldots, L_m)$ is admissible and $\ell_k \in L_j$, (13) can then be rewritten as $\gamma(s \sharp s_1 \sharp I_k) \sqsubseteq \gamma(s \sharp s_2)$. From the definition of the functional $F^\sharp$ we get $s_1 \sharp I_k \sqsubseteq s_2$. The required then follows from the monotonicity of $\sharp$ and γ.

Case 3. C is **release**(ℓ_k). We can assume that $\ell_k \notin L_j$, otherwise the left-hand side of (9) is $\bot$. Thus, we have to show that

$$\tilde{q}(v_1, \ldots, v_j^0, \ldots, v_m, L_1, \ldots, L_j \cup \{\ell_k\}, \ldots, L_m) \sqsubseteq \tilde{q}(v_1, \ldots, v_j, \ldots, v_m, L_1, \ldots, L_j, \ldots, L_m) \quad (14)$$

and

$$\tilde{q}(v_1, \ldots, v_j^0, \ldots, v_m, L_1, \ldots, L_j, \ldots, L_m) \sqsubseteq \tilde{q}(v_1, \ldots, v_j, \ldots, v_m, L_1, \ldots, L_j, \ldots, L_m). \quad (15)$$

We first prove (14). Let s be defined by (11) with $L = L_1 \cup \ldots \cup L_m \cup \{\ell_k\}$, $s_1 = Q(v_j^0, L_j \cup \{\ell_k\})$ and $s_2 = Q(v_j, L_j)$. We can assume that the lockset $(L_1, \ldots, L_j \cup \{\ell_k\}, \ldots, L_m)$ is admissible as otherwise the left-hand side of (14) is $\bot$. Since $\ell_k \notin L_j$, (14) is then equivalent to $\gamma(s \sharp s_1) \sqsubseteq \gamma(s \sharp s_2 \sharp I_k)$. From the definition of the functional $F^\sharp$ we get $\mathsf{Local}_k(s_1) \sqsubseteq s_2$ and $\mathsf{Frame}_k(s_1) \sqsubseteq I_k$. From the monotonicity of $\sharp$ it follows that

$$\mathsf{Local}_k(s_1) \sharp \mathsf{Frame}_k(s_1) \sqsubseteq s_2 \sharp I_k. \quad (16)$$

Then, since $*$ is monotone,

$$\begin{aligned} \gamma(s \sharp s_1) &= \gamma(s) * \gamma(s_1) && \text{by (3)} \\ &\sqsubseteq \gamma(s) * \gamma(\mathsf{Local}_k(s_1) \sharp \mathsf{Frame}_k(s_1)) && \text{by (6)} \\ &\sqsubseteq \gamma(s) * \gamma(s_2 \sharp I_k) && \text{by (16)} \\ &= \gamma(s \sharp s_2 \sharp I_k) && \text{by (3)} \end{aligned}$$

which proves (14). We now proceed to prove (15). Let s be defined by (11) and let $s_1 = Q(v_j^0, L_j)$, $s_2 = Q(v_j, L_j)$. Then (15) is equivalent to $\gamma(s \sharp s_1) \sqsubseteq \gamma(s \sharp s_2)$. From the definition of the functional $F^\sharp$ we have that $s_1 \sqsubseteq s_2$. The required then follows from the monotonicity of γ and $\sharp$.

So, in all the cases (9) is fulfilled, which implies the statement of the theorem. □

B. Proof of Theorem 2 (Data race freedom)

Suppose the contrary: there exist a location $(v_1, \ldots, v_m)$, an admissible lockset $(L_1, \ldots, L_m)$, a state $(t, h) \in q(v_1, \ldots, v_m, L_1, \ldots, L_m)$, CFG edges $(v_1, C_1, v_1') \in E_i$ and $(v_2, C_1, v_2') \in E_j$ $(i \ne j)$ labeled with sequential commands C_1 and C_2 such that $C_1, t, h \not\rightsquigarrow \top$, $C_2, t, h \not\rightsquigarrow \top$ and $C_1 \bowtie_{(t,h)} C_2$. Let $s_1 = Q^\sharp(v_i, L_i)$, $s_2 = Q^\sharp(v_j, L_j)$, and

$$s_0 = \left(\bigsharp_{\substack{1 \le k \le m, \\ k \ne i,\ k \ne j}} Q^\sharp(v_k, L_k)\right) \sharp \left(\bigsharp_{\ell_k \notin L_1 \cup \ldots \cup L_m} I_k^\sharp\right).$$

Then by Theorem 1 and (3), $(t, h) \in \gamma(s_0) * \gamma(s_1) * \gamma(s_2)$. Hence,

$$(t, h) = (t_0, h_0) * (t_1, h_1) * (t_2, h_2), \quad (17)$$

where

$$(t_0, h_0) \in \gamma(s_0), \ \ (t_1, h_1) \in \gamma(s_1), \ \ (t_2, h_2) \in \gamma(s_2). \quad (18)$$

Since $(Q^\sharp, I^\sharp) \sqsubset \top$, from the definition of the functional $F^\sharp$ (Figure 5) it follows that

$$f_{C_1}^\sharp(s_1) \sqsubset \top, \ \ f_{C_2}^\sharp(s_2) \sqsubset \top. \quad (19)$$

Therefore,

$$\begin{aligned} f_{C_1}(t_1, h_1) &\sqsubseteq f_{C_1}(\gamma(s_1)) && \text{by (18)} \\ &\sqsubseteq \gamma(f_{C_1}^\sharp(s_1)) && \text{by (4)} \\ &\sqsubset \top && \text{by (8) and (19)} \end{aligned}$$

So, $f_{C_1}(t_1, h_1) \sqsubset \top$ and, analogously, $f_{C_2}(t_2, h_2) \sqsubset \top$. Hence, $C_1, t_1, h_1 \not\rightsquigarrow \top$ and $C_2, t_2, h_2 \not\rightsquigarrow \top$. From this and the fact that $C_1 \bowtie_{(t,h)} C_2$ using the definition of $*$ and transfer functions for sequential commands given in Section 4.1 we easily get that $(t_1, h_1) * (t_2, h_2)$ is undefined, which contradicts (17). The intuition behind this is that from $C_1, t_1, h_1 \not\rightsquigarrow \top$ and $C_2, t_2, h_2 \not\rightsquigarrow \top$ it follows that both (t_1, h_1) and (t_2, h_2) should have the full permission for the same variable or location accessed by C_1 and C_2, which makes the state $(t_1, h_1) * (t_2, h_2)$ inconsistent. □