Local Reasoning for Storable Locks and Threads

Alexey Gotsman[*] Josh Berdine[†] Byron Cook[†]
Noam Rinetzky[‡] Mooly Sagiv[‡†]

[*]University of Cambridge [†]Microsoft Research
[‡]Tel-Aviv University

April, revised September
Technical Report
MSR-TR-2007-39

We present a resource oriented program logic that is able to reason about concurrent heap-manipulating programs with unbounded numbers of dynamically-allocated locks and threads. The logic is inspired by concurrent separation logic, but handles these more realistic concurrency primitives. We demonstrate that the proposed logic allows local reasoning about programs for which there exists a notion of dynamic ownership of heap parts by locks and threads.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

We are interested in modular reasoning, both manual and automatic, about concurrent heap-manipulating programs. Striking progress in this realm has recently been made by O'Hearn [11], who proposed concurrent separation logic as a basis for reasoning about such programs. Concurrent separation logic is a Hoare logic with two novel features: the assertion language of the logic contains the $*$ connective that splits the program state into disjoint parts, and the proof system has two important rules:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \textsc{Frame} \qquad \frac{\{P_1\}\ C_1\ \{Q_1\} \quad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \parallel C_2\ \{Q_1 * Q_2\}}\ \textsc{Par}$$

According to the FRAME rule, if P includes the part of the program state that C accesses, then executing C in the presence of additional program state R results in the same behavior, and C does not touch the extra state. The PAR rule says that if two processes access disjoint parts of the program state, they can safely execute in parallel and the final state is given by the $*$-conjunction of the postconditions of the processes. Therefore, to reason about a command (or a process) in a program, it is sufficient to consider only the part of the program state that the command actually accesses, a feature that greatly simplifies program proofs and is referred to as the principle of *local reasoning* [10].

In the PAR rule it is intended that the processes access a finite set of shared resources using conditional critical regions to synchronize access. Process interaction is mediated in the logic by assigning to every resource an assertion – its *resource invariant* – that describes the part of the heap owned by the resource and must be respected by every process. For any given process, resource invariants restrict how other processes can interfere with it, and hence, the process can be reasoned about in isolation. In this way the logic allows local reasoning about programs consistent with what O'Hearn terms the Ownership Hypothesis ("A code fragment can access only those portions of state that it owns.") [11], i.e., programs that admit a notion of ownership of heap parts by processes and resources. At the same time, the ownership relation is not required to be static, i.e., it permits ownership transfer of heap cells between areas owned by different processes and resources. The resource-oriented flavor of the logic makes it possible to use it as a basis for thread-modular program analysis [7]: certain classes of resource invariants can automatically be inferred by an abstract interpretation that analyzes each process separately in contrast to a straightforward analysis that just enumerates all execution interleavings.

However, concurrent separation logic [11], its derivatives [1, 14, 4], and a corresponding program analysis [7] all suffer from a common limitation: they assume a bounded number of non-aliased and pre-allocated locks (resources) and threads (processes) and, hence, cannot be used to reason about concurrency primitives present in modern languages and libraries (e.g., POSIX threads) that use unbounded numbers of *storable* locks and threads. Here "storable" means that locks can be dynamically allocated and destroyed in the heap; threads can be dynamically created and can terminate themselves, and moreover, thread identifiers can be stored and subsequently used to wait for termination of the identified thread.

Reasoning about storable locks is especially difficult. The issue here is not that of expressiveness, but of modularity: storable locks can be handled by building a global invariant describing the shared memory as a whole, with all locks allocated in it. However, in this case the locality of reasoning is lost, which kicks back in global invariants containing lots of auxiliary state, proofs being extremely complex and program analyses for discovery of global invariants being infeasible. Recent efforts towards making proofs in this style of reasoning modular [5, 17] use rely-guarantee reasoning to simplify the description of the global invariant and its possible changes (see Section 9 for a detailed comparison of such techniques with our work).

What we want is a logic that preserves concurrent separation logic's local reasoning, even for programs that manipulate storable locks and threads. To this end, in this paper we propose a logic (Section 3), based upon separation logic, that treats storable locks along with the data structures they protect as resources, assigning invariants to them and managing their dynamic creation and destruction. The challenges of designing such a logic were (quite emotionally) summarized by Bornat et al. in [1]:

> ...the idea of semaphores in the heap makes theoreticians wince. The semaphore has to be available to a shared resource bundle:[1] that means a bundle will contain a bundle which contains resource, a notion which makes everybody's eyes water. None of it seems impossible, but it's a significant problem, and solving it will be a small triumph.

Less emotionally, stored locks are analogous to stored procedures in that, unless one is very careful, they can raise a form of Russell's paradox, circularity arising from what Landin called knots in the store. Stored locks can do this by referring to themselves through their resource invariants, and here we address this foundational difficulty by cutting the knots in the store with an indirection.

Our approach to reasoning about storable locks is to represent a lock in the assertion language by a *handle* whose denotation cuts knots in the store. A handle certifies that a lock allocated at a certain address exists and gives a thread owning the handle a permission to (try to) acquire the lock. By using the mechanism of permissions [1] the handle can be split among several threads that can then compete for the lock. Furthermore, a handle carries some information about the part of the program state protected by the lock (its resource invariant), which lets us mediate the interaction among threads, just as in the original concurrent separation logic. Handles for locks can be stored inside resource invariants, thereby permitting reasoning about the situation described in the quote above. In this way we extend the ability of concurrent separation logic to reason locally about programs that are consistent with the Ownership Hypothesis to the setting with storable locks and threads. As we show in Section 4, the class of

[1] Here the term "resource bundle" is used to name what we, following O'Hearn's original paper, call "resource invariant".

such programs contains programs with coarse-grained synchronization and some, but not all, programs with fine-grained synchronization, including examples that were posed as challenges in the literature.

We prove the logic sound with respect to an interleaving operational semantics (Section 7). It happens that even formulating the soundness statement is non-trivial as we have to take into account resource invariants for locks not mentioned directly in the local states of threads.

The technical issues involved in reasoning about storable locks and storable threads are similar. To make the presentation more approachable, we first present a logic for programs consisting of one top-level parallel composition of several threads. In Section 8 we extend the logic to handle dynamic thread creation.

2 Technical Background

In this section we review some technical concepts of (sequential) separation logic that we reuse in ours. We consider a version of separation logic that is a Hoare logic for a heap-manipulating programming language with the following syntax:

V	$::= l, x, y, \ldots$	variables
E, F	$::= \mathsf{nil} \mid V \mid E + F \mid \ \ldots$	expressions
G	$::= E = F \mid E \neq F$	branch guards
C	$::= V = E \mid V = [E] \mid [E] = F \mid V = \textbf{new} \mid \textbf{delete}\ E$	primitive commands
S	$::= C \mid S; S \mid \textbf{if}\ G\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{fi} \mid \textbf{while}\ G\ \textbf{do}\ S\ \textbf{od}$	commands

Here square brackets denote pointer dereferencing; the meaning of the rest of the language is standard.

Formulae in the assertion language of separation logic denote program states represented by stack-heap pairs and have the following syntax:

$$\begin{aligned}\Phi ::= {} & \mathsf{false} \mid \Phi \Rightarrow \Phi \mid \exists X.\Phi \mid \Phi * \Phi \mid \Phi -\!\!* \ \Phi \mid \mathsf{emp_s} \mid \mathsf{emp_h} \\ \mid {} & E = F \mid \pi = \mu \mid \mathsf{Own}_\pi(x) \mid E \mapsto F\end{aligned}$$

We can define usual connectives not mentioned in the syntax definition using the provided ones. Note that we treat variables as resources [14] to avoid side conditions in proof rules, i.e., we treat the stack in the same way as the heap, Thus, the assertion $E \mapsto F$ denotes the set of stack-heap pairs such that the heap consists of one cell allocated at the address E and storing the value F, and the stack contains all variables mentioned in E and F. The assertion $\mathsf{Own}_1(x)$ (the general form $\mathsf{Own}_\pi(x)$ is explained later) restricts the stack to contain only the variable x and leaves the heap unconstrained. We can separate assertions about variable ownership $\mathsf{Own}_1(x)$ with $*$ in the same way as assertions $E \mapsto F$ about ownership of heap cells. $\mathsf{emp_s}$ describes the empty stack and $\mathsf{emp_h}$ the empty heap. We distinguish integer program variables $x, y, \ldots$ (which may appear in programs) and logical variables $X, Y, \ldots$ (which do not appear in programs, only in formulae). In the assertion language definition E and F range over expressions, which are the same as in the programming language, but can contain logical variables. We write $E \mapsto \_$ for $\exists X. E \mapsto X$ where X does not occur free in E.

The assertion language includes fractional permissions [1] for variables, which are necessary for getting a complete (in the sense of [14]) proof system when variables are treated as resources. For clarity of presentation we omit the treatment of permissions for heap cells. Permissions are denoted with permission expressions (ranged over by π and μ), which are expressions evaluating to numbers from $(0,1]$. A permission shows "how much" of a variable is owned by the assertion. For example, variable x represented by $\mathsf{Own}_1(x)$ can be split into two permissions $\mathsf{Own}_{1/2}(x)$, each of which permits reading the variable, but not writing to it. Two permissions $\mathsf{Own}_{1/2}(x)$ can later be recombined to obtain the full permission $\mathsf{Own}_1(x)$, which allows both reading from and writing to x. We make the convention that $\Vdash$ binds most loosely, use $\pi_1 x_1, \ldots, \pi_k x_k \Vdash P$ to denote $\mathsf{Own}_{\pi_1}(x_1) * \ldots * \mathsf{Own}_{\pi_k}(x_k) \wedge P$ and abbreviate $1x$ to x.

The proof rules (see Appendix A) are the same as in [15, 14] modulo treating variables as resources in heap-manipulating commands. In the rules and the following, O ranges over assertions of the form $\pi_1 x_1, \ldots, \pi_k x_k$. We also allow O to be empty, in which case we interpret $O \Vdash P$ as $\mathsf{emp_s} \wedge P$.

3 Logic

We now consider a concurrent programming language based on the sequential one presented in Section 2:

$$\begin{array}{ll} C ::= \ldots \mid \mathbf{init}(E) \mid \mathbf{finalize}(E) \mid \mathbf{acquire}(E) \mid \mathbf{release}(E) & \text{primitive commands} \\ P ::= S \parallel \ldots \parallel S & \text{programs} \end{array}$$

We assume that each program consists of one parallel composition of several threads. Synchronization is performed using locks, which are dynamically created and destroyed in the heap. $\mathbf{init}(E)$ converts a location allocated at the address E to a lock. After the completion of $\mathbf{init}(E)$ the thread that executed it holds the lock. $\mathbf{acquire}(E)$ and $\mathbf{release}(E)$ try to acquire, respectively, release the lock allocated at the address E. $\mathbf{finalize}(E)$ converts the lock into an ordinary heap cell containing an unspecified value provided that the lock at the address E is held by the thread that is executing the command.

As in concurrent separation logic [11], with each lock we associate a resource invariant – a formula that describes the part of the heap protected by the lock. (This association is considered to be part of the proof, rather than of the program.) To deal with unbounded numbers of locks we assume that each lock has a *sort* that determines its invariant. Formally, we assume a fixed set $\mathcal{L}$ of function symbols with positive arities representing lock sorts, and with each $A \in \mathcal{L}$ of arity k we associate a formula $I_A(L, \vec{X})$ containing k free logical variables specified as parameters – the resource invariant for the sort A. The meaning of the first parameter is fixed as the address at which the lock is allocated. Other parameters can have arbitrary meaning. In Sections 5 and 7 we give certain restrictions that resource invariant formulae must satisfy for the logic to be sound.

We extend the assertion language of separation logic with two extra forms: $\Phi ::= \ldots \mid \pi A(E, \vec{F}) \mid \mathsf{Locked}_A(E, \vec{F})$. An expression of the form $A(E, \vec{F})$, where

$$\frac{(O \Vdash E{\mapsto}\_) \Rightarrow \vec{F} = \vec{F}}{\{O \Vdash E{\mapsto}\_\}\ \mathbf{init}_{A,\vec{F}}(E)\ \{O \Vdash A(E,\vec{F}) * \mathsf{Locked}_A(E,\vec{F})\}}\ \textsc{Init}$$

$$\frac{}{\{O \Vdash A(E,\vec{F}) * \mathsf{Locked}_A(E,\vec{F})\}\ \mathbf{finalize}(E)\ \{O \Vdash E{\mapsto}\_\}}\ \textsc{Finalize}$$

$$\frac{}{\{(O \Vdash \pi A(L,\vec{X})) \wedge L{=}E\}\ \mathbf{acquire}(E)\ \{(O \Vdash \pi A(L,\vec{X}) * \mathsf{Locked}_A(L,\vec{X})) * I_A(L,\vec{X})\}}\ \textsc{Acquire}$$

$$\frac{}{\{((O \Vdash \mathsf{Locked}_A(L,\vec{X})) * I_A(L,\vec{X})) \wedge L{=}E\}\ \mathbf{release}(E)\ \{O \Vdash \mathsf{emp}_{\mathbf{h}}\}}\ \textsc{Release}$$

Fig. 1. Proof rules for lock-manipulating commands

$A \in \mathcal{L}$, is a *handle* for the lock of the sort A allocated at the address E. It can be viewed as an existential permission for the lock: a thread having $A(E,\vec{F})$ knows that the heap cell at the address E is allocated and is a lock, and can try to acquire it. $A(E,\vec{F})$ does not give permissions for reading from or writing to the cell at the address E. Moreover, it does not ensure that the part of the heap protected by the lock satisfies the resource invariant until the thread successfully acquires the lock. We allow using $A(E,\vec{F})$ with fractional permissions [1] writing $\pi A(E,\vec{F})$. The intuition behind the permissions is that a handle for a lock with the full permission 1 can be split among several threads, thereby allowing them to compete for the lock. A thread having a permission for the handle less than 1 can acquire the lock; a thread having the full permission can in addition finalize the lock. We abbreviate $1A(E,\vec{F})$ to $A(E,\vec{F})$. Assertions in the code of threads can also use a special form $\mathsf{Locked}_A(E,\vec{F})$ to represent the fact that the lock at the address E is held by the thread in the surrounding code of the assertion. $\mathsf{Locked}_A(E,\vec{F})$ also ensures that the cell at the address E is allocated and is a lock of the sort A with parameters $\vec{F}$.

Our logic includes the proof rules of sequential separation logic and four new rules for lock-manipulating commands shown in Figure 1. We do not provide a rule for parallel composition as our programs consist of only one top-level parallel composition, and in Section 8 we instead treat dynamic thread creation. We write $\vdash \{P\}\ C\ \{Q\}$ to denote that the triple $\{P\}\ C\ \{Q\}$ is provable in our logic.

Initializing a lock (Init) converts a cell in the heap at the address E to a lock. Upon completion of $\mathbf{init}(E)$ the thread that executed it gets both the ownership (with the full permission) of the handle $A(E,\vec{F})$ for the lock and the knowledge that it holds the lock, represented by $\mathsf{Locked}_A(E,\vec{F})$. Note that for the precondition $O \Vdash E{\mapsto}\_$ to be consistent O must contain variables mentioned in E. In this and other rules we use O to supply the permissions for variables necessary for executing the command. For $\mathbf{init}_{A,\vec{F}}(E)$ commands to be safe the stack must contain variables mentioned in E and $\vec{F}$, hence, the premiss $(O \Vdash E{\mapsto}\_) \Rightarrow \vec{F} = \vec{F}$ additionally requires that variables be contained in O (see [14]). An implicit side condition in the Init rule is that in all branches of a proof of a program, the sort A of the lock and the values of parameters $\vec{F}$ have to be chosen consistently for each **init** command (as otherwise the conjunction rule of Hoare logic becomes unsound). This is formally enforced by annotating each **init** command with the sort of the lock that is being created and its parameters (defined by arbitrary expressions $\vec{F}$ over program variables). In general, the lock sort can also be computed as a function of program variables. To simplify notation we assume the sort of the lock is fixed for each **init** command in the program. We note that although we use the sort of the lock and its parameters for conceptually different purposes (see the examples in Section 4), technically they are merely pieces of auxiliary state associated with the handle for the lock that carry some information about the resource invariant of the lock. Therefore, the annotations of lock sorts and parameters at **init** commands can be viewed as just assignments to auxiliary cells in memory.

Finalizing a lock results in it being converted into an ordinary cell. To finalize a lock (Finalize) a thread has to have the full permission for the handle $A(E, \vec{F})$ associated with the lock. Additionally, the lock has to be held by the thread, i.e., $\mathsf{Locked}_A(E, \vec{F})$ has to be in its local state.

A thread can acquire a lock if it has a permission for the handle of the lock. Acquiring a lock (Acquire) results in the resource invariant of the lock (with appropriately instantiated parameters) being $*$-conjoined to the local state of the thread. The thread also obtains the corresponding Locked fact, which guarantees that it holds the lock. A thread acquiring the same lock twice deadlocks, which is enforced by $\mathsf{Locked}_A(E, \vec{F}) * \mathsf{Locked}_A(E, \vec{F})$ being inconsistent (see Section 5). Conversely, a thread can release a lock (Release) only if it holds the lock, i.e., the corresponding Locked fact is present in the local state of the thread. Upon releasing the lock the thread gives up both this knowledge and the ownership of the resource invariant associated with the lock. The fact that resource invariants can claim ownership of program variables complicates the rules Acquire and Release. E.g., in the postcondition of Acquire we cannot put $I_A(L, \vec{X})$ inside the expression after $\Vdash$ as it may claim ownership of variables not mentioned in O. This requires us to use a logical variable L in places where the expression E would have been expected.

4 Examples of Reasoning

We first show (in Example 1 below) that straightforward application of rules for lock-manipulating commands allows us to handle programs in which locks protect parts of the heap without other locks allocated in them. We then present two more involved examples of using the logic, which demonstrate how extending the logic with storable locks has enabled reasoning more locally than was previously possible in some interesting cases (Examples 2 and 3).

Instead of the minimalistic language presented in Section 3, in our examples we use a language with some additional C-like syntax (in particular, C structures) that can easily be desugared to the language of Section 3. For an address x of a structure, we use $x.F$ in the assertion language as syntactic sugar for $x + d$,

```
struct RECORD {
  LOCK Lock;
  int Data;
};

main() {
  RECORD *x;
  {x ⊩ emp_h}
  x = new RECORD;
  {x ⊩ x↦_ ∗ x.Data↦_}
  init_R(x);
  {x ⊩ x.Data↦_ ∗ R(x) ∗
   Locked_R(x)}
```

```
  x->Data = 0;
  {x ⊩ x.Data↦0 ∗ R(x) ∗
   Locked_R(x)}
  release(x);
  {x ⊩ R(x)}
  // ...
  {x ⊩ R(x)}
  acquire(x);
  {x ⊩ x.Data↦_ ∗ R(x) ∗
   Locked_R(x)}
  x->Data++;
  {x ⊩ x.Data↦_ ∗ R(x) ∗
   Locked_R(x)}
```

```
  release(x);
  {x ⊩ R(x)}
  // ...
  {x ⊩ R(x)}
  acquire(x);
  {x ⊩ x.Data↦_ ∗ R(x) ∗
   Locked_R(x)}
  finalize(x);
  {x ⊩ x↦_ ∗ x.Data↦_}
  delete x;
  {x ⊩ emp_h}
}
```

$$I_R(L) \stackrel{\Delta}{=} \mathsf{emp_s} \wedge L.Data \mapsto \_$$

Fig. 2. A very simple example of reasoning in the logic

where d is the offset of the field F in the structure. We assume that each field in a structure takes one memory cell. We also use an obvious generalization of **new** and **delete** that allocate and deallocate several memory cells at once.

Example 1: A simple situation. Figure 2 shows a proof outline for a program with a common pattern: a lock-field in a structure protecting another field in the same structure. We use a lock sort R with invariant $I_R(L)$. The proof outline shows how the "life cycle" of a lock is handled in our proof system: creating a cell, converting it to a lock, acquiring and releasing the lock, converting it to an ordinary cell, and disposing the cell. For simplicity we consider a program with only one thread.

Example 2: "Last one disposes". This example was posed as a challenge for local reasoning in [1]. The program in Figure 3 represents a piece of multicasting code: a single packet p (of type *PACKET*) with *Data* inside the packet is distributed to M *thread*s at once. For efficiency reasons instead of copying the packet, it is shared among threads. A *Count* of access permissions protected by *Lock* is used to determine when everybody has finished and the packet can be disposed. The program consists of a top level parallel composition of M calls to the procedure *thread*. Here M is a constant assumed to be greater than 0. For completeness, we also provide the procedure *initialize* that can be used to initialize the packet and thereby establish the precondition of the program.

To prove the program correct the resource invariant for the lock at the address p has to contain a partial permission for the handle of *the same lock*. This is formally represented by a lock sort P with the resource invariant $I_P(L, M)$. Initially the resource invariant contains no permissions of this kind and the handle $P(p, M)$ for the lock is split among M threads (hence, the precondition of each thread is $(1/M)p \Vdash (1/M)P(p, M)$). Each thread uses the handle to acquire the lock and process the packet. When a thread finishes processing and releases the lock, it transfers the permission for the handle it owned to the resource invariant of the lock. The last thread to process the packet can then

```
struct PACKET { LOCK Lock;
                int Count;
                DATA Data; };
PACKET *p;

thread() {
  ${(1/M)p \Vdash (1/M)P(p, M)}$
  acquire(p);
  ${(1/M)p \Vdash \exists X.0 \leq X < M \wedge p.Count \mapsto X * p.Data \mapsto \_ * ((X + 1)/M)P(p, M) * \mathsf{Locked}_P(p, M)}$
  // ...Process data...
  p->Count++;
  ${(1/M)p \Vdash \exists X.1 \leq X \leq M \wedge p.Count \mapsto X * p.Data \mapsto \_ * (X/M)P(p, M) * \mathsf{Locked}_P(p, M)}$
  if (p->Count == M) {
    ${(1/M)p \Vdash p.Count \mapsto M * p.Data \mapsto \_ * P(p, M) * \mathsf{Locked}_P(p, M)}$
    // ...Finalize data...
    finalize(p);
    ${(1/M)p \Vdash p.Count \mapsto M * p.Data \mapsto \_ * p \mapsto \_}$
    delete(p);
  } else {
    ${(1/M)p \Vdash \exists X.1 \leq X < M \wedge p.Count \mapsto X * p.Data \mapsto \_ * (X/M)P(p, M) * \mathsf{Locked}_P(p, M)}$
    release(p);
  }
  ${(1/M)p \Vdash \mathsf{emp}_{\mathbf{h}}}$
}

initialize() {
  ${p \Vdash \mathsf{emp}_{\mathbf{h}}}$
  p = new PACKET;
  ${p \Vdash p \mapsto \_ * p.Count \mapsto \_ * p.Data \mapsto \_}$
  p->Count = 0;
  ${p \Vdash p \mapsto \_ * p.Count \mapsto 0 * p.Data \mapsto \_}$
  init(p);
  // ...Initialize data...
  ${p \Vdash p.Count \mapsto 0 * p.Data \mapsto \_ * P(p, M) * \mathsf{Locked}_P(p, M)}$
  release(p);
  ${p \Vdash P(p, M)}$
}
```

Fig. 3. Proof outline for the "Last one disposes" program

get the full permission for the lock by combining the permission in the invariant with its own one and can therefore dispose the packet.

Example 3: Lock coupling list. We next consider a fine-grained implementation of a singly-linked list with concurrent access, whose nodes store integer keys. The program (Figures 4 and 5) consists of M operations *add* and *remove* running in parallel. The operations add and remove an element with the given key to or from the list. Traversing the list uses lock coupling: the lock on one node is not released until the next node is locked. The list is sorted and the first and last nodes in it are sentinel nodes that have values $-\infty$, respectively, $+\infty$. It is initialized by the code in procedure *initialize*. We only provide a proof outline for the procedure *locate* (Figure 4), which is invoked by other procedures to traverse the list. We use lock sorts H (for the head node) and N (for all other nodes) with the invariants $I_H(L)$ and $I_N(L, V)$. In this example the resource invariant for the lock protecting a node in the list holds a handle for the lock protecting the next node in the list. The full permission for $N(X, V')$ in the invariants above essentially means that the only way a thread can lock a node is by first locking its predecessor: here the invariant enforces a particular locking policy.

```
locate(int e) {
  NODE *prev, *curr;
  $\{O \Vdash -\infty < e \wedge (1/M)H(head)\}$
  prev = head;
  $\{O \Vdash -\infty < e \wedge prev = head \wedge (1/M)H(head)\}$
  acquire(prev);
  $\{O \Vdash \exists V'.{-\infty} < e \wedge -\infty < V' \wedge (1/M)H(head) * \mathsf{Locked}_H(prev) *$
   $\exists X.prev.Val{\mapsto}{-\infty} * prev.Next{\mapsto}X * N(X, V')\}$
  curr = prev->Next;
  $\{O \Vdash \exists V'.{-\infty} < e \wedge -\infty < V' \wedge (1/M)H(head) * \mathsf{Locked}_H(prev) *$
   $prev.Val{\mapsto}{-\infty} * prev.Next{\mapsto}curr * N(curr, V')\}$
  acquire(curr);
  $\{O \Vdash \exists V'.{-\infty} < e \wedge -\infty < V' \wedge (1/M)H(head) * N(curr, V') * \mathsf{Locked}_H(prev) *$
   $\mathsf{Locked}_N(curr, V') * prev.Val{\mapsto}{-\infty} * prev.Next{\mapsto}curr * curr.Val{\mapsto}V' *$
   $((curr.Next{\mapsto}\mathsf{nil} \wedge V' = +\infty) \vee (\exists X, V''.curr.Next{\mapsto}X * N(X, V'') \wedge V' < V''))\}$
  while (curr->Val < e) {
    $\{O \Vdash \exists V, V'.V' < e \wedge (1/M)H(head) * N(curr, V') * \mathsf{Locked}_N(curr, V') *$
     $(\mathsf{Locked}_H(prev) \wedge V = -\infty \vee \mathsf{Locked}_N(prev, V)) * prev.Val{\mapsto}V * prev.Next{\mapsto}curr *$
     $\exists X, V''.curr.Val{\mapsto}V' * curr.Next{\mapsto}X * N(X, V'') \wedge V < V' < V''\}$
    release(prev);
    $\{O \Vdash \exists X, V', V''.V' < e \wedge V' < V'' \wedge (1/M)H(head) * \mathsf{Locked}_N(curr, V') *$
     $curr.Val{\mapsto}V' * curr.Next{\mapsto}X * N(X, V'')\}$
    prev = curr;
    curr = curr->Next;
    $\{O \Vdash \exists V, V'.V < e \wedge V < V' \wedge (1/M)H(head) * \mathsf{Locked}_N(prev, V) *$
     $prev.Val{\mapsto}V * prev.Next{\mapsto}curr * N(curr, V')\}$
    acquire(curr);
    $\{O \Vdash \exists V, V'.V < e \wedge V < V' \wedge (1/M)H(head) * \mathsf{Locked}_N(prev, V) *$
     $\mathsf{Locked}_N(curr, V') * N(curr, V') * prev.Val{\mapsto}V * prev.Next{\mapsto}curr * curr.Val{\mapsto}V' *$
     $((V' = +\infty \wedge curr.Next{\mapsto}\mathsf{nil}) \vee \exists X, V''.curr.Next{\mapsto}X * N(X, V'') \wedge V' < V'')\}$
  }
  $\{O \Vdash \exists V, V'.V < e \leq V' \wedge (1/M)H(head) * \mathsf{Locked}_N(prev, V) * \mathsf{Locked}_N(curr, V') * N(curr, V') *$
   $prev.Val{\mapsto}V * prev.Next{\mapsto}curr * curr.Val{\mapsto}V' * ((V' = +\infty \wedge curr.Next{\mapsto}\mathsf{nil}) \vee$
   $\exists X, V''.curr.Next{\mapsto}X * N(X, V'') \wedge V' < V'')\}$
  return (prev, curr);
}
```

$$
\begin{aligned}
I_H(L) &\stackrel{\Delta}{=} \mathsf{emp_s} \wedge \exists X, V'.L.Val{\mapsto}{-\infty} * L.Next{\mapsto}X * N(X, V') \wedge -\infty < V' \\
I_N(L, V) &\stackrel{\Delta}{=} \mathsf{emp_s} \wedge ((L.Val{\mapsto}V * L.Next{\mapsto}\mathsf{nil} \wedge V = +\infty) \vee \\
&\qquad (\exists X, V'.L.Val{\mapsto}V * L.Next{\mapsto}X * N(X, V') \wedge V < V'))
\end{aligned}
$$

Fig. 4. Proof outline for a part of the lock coupling list program. Here O is e, *prev*, *curr*, $(1/M)head$.

```
struct NODE { LOCK Lock;
              int Val;
              NODE *Next; }

NODE *head;

initialize() {
  NODE *last;
  last = new NODE;
  last->Val = INFINITY;
  last->Next = NULL;
  init_{N,+∞}(last);
  release(last);
  head = new NODE;
  head->Val = -INFINITY;
  head->Next = last;
  init_H(head);
  release(head);
}
```

```
add(int e) {
  NODE *n1, *n2, *n3, *result;
  (n1, n3) = locate(e);
  if (n3->Val != e) {
    n2 = new NODE;
    n2->Val = e;
    n2->Next = n3;
    init_{N,e}(n2);
    release(n2);
    n1->Next = n2;
    result = true;
  } else {
    result = false;
  }
  release(n1);
  release(n3);
  return result;
}
```

```
remove(int e) {
  NODE *n1, *n2, *n3;
  NODE *result;
  (n1, n2) = locate(e);
  if (n2->Val == e) {
    n3 = n2->Next;
    n1->Next = n3;
    finalize(n2);
    delete n2;
    result = true;
  } else {
    release(n2);
    result = false;
  }
  release(n1);
  return result;
}
```

Fig. 5. Lock coupling list program. The procedure *locate* is shown in Figure 4.

$$\begin{array}{ll}
\mathsf{nil} = 0 & \text{Values} = \{\ldots, -1, 0, 1, \ldots\} \\
\text{Perms} = (0,1] & \text{Vars} = \{x, y, \ldots\} \\
\text{Stacks} = \text{Vars} \rightharpoonup_{\text{fin}} (\text{Values} \times \text{Perms}) & \text{Locs} = \{1, 2, \ldots\} \\
\text{LockPerms} = [0,1] & \text{ThreadIDs} = \{1, 2, \ldots\} \\
\text{LockVals} = \{\mathsf{U}, 0\} \cup \text{ThreadIDs} & \text{States} = \text{Stacks} \times \text{Heaps}
\end{array}$$

$$\begin{array}{l}
\text{Heaps} = \text{Locs} \rightharpoonup_{\text{fin}} (\mathsf{Cell}(\text{Values}) \cup \mathsf{Lock}(\mathcal{L} \times \text{LockVals} \times \text{LockPerms}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \setminus \mathsf{Lock}(\mathcal{L} \times \{\mathsf{U}\} \times \{0\}))
\end{array}$$

Fig. 6. Model of the assertion language

We were able to present modular proofs for the programs above because in each case we could associate with every lock a part of the heap such that a thread accessed the part only when it held the lock, that is, the lock owned the part of the heap. We note that we would not be able to give modular proofs to programs that do not obey this policy, for instance, to optimistic list [16] – another fine-grained implementation of the list from Example 3 in which the procedure *locate* first traverses the list without taking any locks and then validates the result by locking two candidate nodes and re-traversing the list to check that they are still present and adjacent in the list.

5 Model of the Assertion Language

As usual, assertion language formulae denote sets of pairs of a stack and a heap, both represented by finite partial functions. They are interpreted over the domain in Figure 6. However, in contrast to the standard domain used in separation logic, here cells in the heap can be of two types: ordinary cells (Cell) and locks (Lock). A lock has a sort, a value, and is associated with a permission from $[0, 1]$. To simplify notation, here and in the further semantic development we assume that lock sorts have no parameters other than the address of the lock. Our results can straightforwardly be adjusted to the general case (parameters can be treated in

$$
\begin{array}{lll}
(s,h,i) \models_k E \mapsto F & \Leftrightarrow & \llbracket E \rrbracket_{(s,i)}\downarrow \wedge \llbracket F \rrbracket_{(s,i)}\downarrow \wedge h = [\llbracket E \rrbracket_{(s,i)} : \mathsf{Cell}(\llbracket F \rrbracket_{(s,i)})] \\
(s,h,i) \models_k \mathsf{Own}_{\pi}(x) & \Leftrightarrow & \exists u. \llbracket \pi \rrbracket_{(s,i)}\downarrow \wedge s = [x : (u, \llbracket \pi \rrbracket_{(s,i)})] \wedge 0 < \llbracket \pi \rrbracket_{(s,i)} \leq 1 \\
(s,h,i) \models_k \pi A(E) & \Leftrightarrow & \\
\multicolumn{3}{l}{\qquad \llbracket E \rrbracket_{(s,i)}\downarrow \wedge \llbracket \pi \rrbracket_{(s,i)}\downarrow \wedge h = [\llbracket E \rrbracket_{(s,i)} : \mathsf{Lock}(A, \mathsf{U}, \llbracket \pi \rrbracket_{(s,i)})] \wedge 0 < \llbracket \pi \rrbracket_{(s,i)} \leq 1} \\
(s,h,i) \models_k \mathsf{Locked}_A(E) & \Leftrightarrow & \llbracket E \rrbracket_{(s,i)}\downarrow \wedge h = [\llbracket E \rrbracket_{(s,i)} : \mathsf{Lock}(A, k, 0)] \\
(s,h,i) \models_k \mathsf{emp}_{\mathbf{s}} & \Leftrightarrow & s = [\,] \\
(s,h,i) \models_k \mathsf{emp}_{\mathbf{h}} & \Leftrightarrow & h = [\,] \\
(s,h,i) \models_k E = F & \Leftrightarrow & \llbracket E \rrbracket_{(s,i)}\downarrow \wedge \llbracket F \rrbracket_{(s,i)}\downarrow \wedge \llbracket E \rrbracket_{(s,i)} = \llbracket F \rrbracket_{(s,i)} \\
(s,h,i) \models_k \pi = \mu & \Leftrightarrow & \llbracket \pi \rrbracket_{(s,i)}\downarrow \wedge \llbracket \mu \rrbracket_{(s,i)}\downarrow \wedge \llbracket \pi \rrbracket_{(s,i)} = \llbracket \mu \rrbracket_{(s,i)} \\
(s,h,i) \models_k P \Rightarrow Q & \Leftrightarrow & ((s,h,i) \models_k P) \Rightarrow ((s,h,i) \models_k Q) \\
(s,h,i) \models_k \mathsf{false} & \Leftrightarrow & \mathsf{false} \\
(s,h,i) \models_k P * Q & \Leftrightarrow & \\
\multicolumn{3}{l}{\qquad \exists s_1, h_1, s_2, h_2 . s = s_1 * s_2 \wedge h = h_1 * h_2 \wedge (s_1, h_1, i) \models_k P \wedge (s_2, h_2, i) \models_k Q} \\
(s,h,i) \models_k P \mathrel{-\!\!*} Q & \Leftrightarrow & \\
\multicolumn{3}{l}{\qquad \forall s', h' . s \sharp s' \wedge h \sharp h' \wedge ((s', h', i) \models_k P) \Rightarrow ((s * s', h * h', i) \models_k Q)} \\
(s,h,i) \models_k \exists X.P & \Leftrightarrow & \exists u.(s, h, i[X : u]) \models_k P
\end{array}
$$

Fig. 7. Satisfaction relation for the assertion language formulae: $(s,h,i) \models_k \Phi$

the same way as lock sorts). The permission 0 is used to represent the existential permission for a lock that is carried by $\mathsf{Locked}_A(E, \vec{F})$. Locks are interpreted as follows: 0 represents the fact that the lock is not held by any thread (i.e., is *free*), values from ThreadIDs represent the identifier of the thread that holds the lock, and U means that the status of the lock is unknown. U is not encountered in the states obtained in the operational semantics we define in Section 6, but is used for interpreting formulae representing parts of complete states. The semantics of formulae and commands never encounter locks of form $\mathsf{Lock}(A, \mathsf{U}, 0)$ for any A, and so the definition of Heaps removes them in order to make the $*$ operation on states cancellative [4].

Note how Heaps in the domain of Figure 6 is not defined recursively, but instead uses an indirection through $\mathcal{L}$, whose elements are associated with resource invariants, and hence indirectly to Heaps. It is this indirection that deals with the foundational circularity issue raised by locks which may refer to themselves.

In this paper we use the following notation for partial functions: $f(x)\downarrow$ means that the function f is defined on x, $f(x)\uparrow$ that the function f is undefined on x, and $[\,]$ denotes a nowhere-defined function. Furthermore, we denote with $f[x : y]$ (defined only if $f(x)\uparrow$) the function that has the same value as f everywhere, except for x, where it has the value y. We abbreviate $[\,][x : y]$ to $[x : y]$.

We now define $*$ on states in our domain, which interprets the $*$-connective in the logic. We first define the $*$ operation on values of locks in the following way: $\mathsf{U} * \mathsf{U} = \mathsf{U}$, $k * \mathsf{U} = \mathsf{U} * k = k$, and $k * j$ is undefined for $k, j \in \{0\} \cup \text{ThreadIDs}$. Note that $k * k$ is undefined as it arises in the cases when a thread tries to acquire a lock twice (recall that we specify that a thread deadlocks in this case).

For $s_1, s_2 \in$ Stacks let

$$s_1 \sharp s_2 \Leftrightarrow \forall x. s_1(x)\downarrow \wedge s_2(x)\downarrow \Rightarrow (\exists v, \pi_1, \pi_2 . s_1(x) = (v, \pi_1) \wedge s_2 = (v, \pi_2) \wedge \pi_1 + \pi_2 \leq 1) .$$

If $s_1 \sharp s_2$, then

$$s_1 * s_2 = \{(x,(v,\pi)) \mid (s_1(x) = (v,\pi) \wedge s_2(x)\uparrow) \vee (s_2(x) = (v,\pi) \wedge s_1(x)\uparrow) \vee \\ (s_1(x) = (v,\pi_1) \wedge s_2(x) = (v,\pi_2) \wedge \pi = \pi_1 + \pi_2)\} \ ,$$

otherwise $s_1 * s_2$ is undefined. For $h_1, h_2 \in$ Heaps let

$$h_1 \sharp h_2 \Leftrightarrow \forall u. h_1(u){\downarrow} \wedge h_2(u){\downarrow} \Rightarrow ((\exists v. h_1(u) = h_2(u) = \mathsf{Cell}(v)) \vee (\exists A, v_1, v_2, \pi_1, \pi_2. \\ h_1(u) = \mathsf{Lock}(A, v_1, \pi_1) \wedge h_2(u) = \mathsf{Lock}(A, v_2, \pi_2) \wedge v_1 * v_2{\downarrow} \wedge \pi_1 + \pi_2 \leq 1)) \ .$$

If $h_1 \sharp h_2$, then

$$h_1 * h_2 = \{(u, \mathsf{Cell}(v)) \mid h_1(u) = \mathsf{Cell}(v) \vee h_2(u) = \mathsf{Cell}(v)\} \cup \\ \{(u, \mathsf{Lock}(A, v, \pi)) \mid (h_1(u) = \mathsf{Lock}(A, v, \pi) \wedge h_2(u)\uparrow) \vee (h_2(u) = \mathsf{Lock}(A, v, \pi) \wedge h_1(u)\uparrow) \\ \vee (h_1(u) = \mathsf{Lock}(A, v_1, \pi_1) \wedge h_2(u) = \mathsf{Lock}(A, v_2, \pi_2) \wedge \pi = \pi_1 + \pi_2 \wedge v = v_1 * v_2)\} \ ,$$

otherwise $h_1 * h_2$ is undefined. We lift $*$ to states and sets of states pointwise.

The satisfaction relation for the assertion language formulae is defined in Figure 7. A formula is interpreted with respect to a thread identifier $k \in \{0\} \cup$ ThreadIDs, a stack s, a heap h, and an interpretation i mapping logical variables to Values. Note that in this case it is convenient for us to consider 0 as a dummy thread identifier. We assume a function $[\![E]\!]_{(s,i)}$ that evaluates an expression with respect to the stack s and the interpretation i. We consider only interpretations that define the value of every logical variable used. We omit i when s suffices to evaluate the expression. We let $[\![P]\!]_i^k$ denote the set of states in which the formula P is valid with respect to the thread identifier k and the interpretation i and let $\mathcal{I}_k(A, u) = [\![I_A(L) * \mathsf{Locked}_A(L)]\!]_{[L:u]}^k$.

We say that a predicate $p \subseteq$ States is *precise* [11] if for any state σ, there exists at most one substate σ_0 (i.e., $\sigma = \sigma_0 * \sigma_1$ for some σ_1) satisfying p. We say that a predicate p is *intuitionistic* [9] if it is closed under stack and heap extension: if p is true of a state σ_1, then for any state σ_2, such that $\sigma_1 * \sigma_2$ is defined, p is also true of $\sigma_1 * \sigma_2$. We say that a predicate p *has an empty lockset* if the value of any lock in every state satisfying p is U. A formula is precise, intuitionistic, or has an empty lockset if its denotation with respect to any thread identifier and interpretation of logical variables is precise, intuitionistic, or has an empty lockset. We require that formulae representing resource invariants be precise and have an empty lockset, i.e., that for each u and k the predicate $[\![I_A(L)]\!]_{[L:u]}^k$ be precise and have an empty lockset. The former requirement is inherited from concurrent separation logic, where it is required for soundness of the conjunction rule. The latter requirement is necessary for soundness of our logic and stems from the fact that in our semantics we do not allow a thread that did not acquire a lock to release it (in agreement with the semantics of mutexes in the POSIX threads library). If we were to allow this (i.e., if we treated locks as binary semaphores rather than mutexes), then this requirement would not be necessary. It is easy to check that the invariants for lock sorts R, P, H, and N from Section 4 satisfy these constraints.

$$\begin{array}{ll}
x = E, (s[x : (u,1)], h) & \rightsquigarrow_k (s[x : (\llbracket E \rrbracket_{s[x:(u,1)]}, 1)], h) \\
x = [E], (s[x : (u,1)], h[e : \mathsf{Cell}(v)]) & \rightsquigarrow_k (s[x : (v,1)], h[e : \mathsf{Cell}(v)]), e = \llbracket E \rrbracket_{s[x:(u,1)]} \\
[E] = F, (s, h[\llbracket E \rrbracket_s : \mathsf{Cell}(u)]) & \rightsquigarrow_k (s, h[\llbracket E \rrbracket_s : \mathsf{Cell}(\llbracket F \rrbracket_s)]) \\
x = \mathbf{new}, (s[x : (u,1)], h) & \rightsquigarrow_k (s[x : (v,1)], h[v : \mathsf{Cell}(w)]),\ \text{if } h(v)\uparrow \\
\mathbf{delete}\ E, (s, h[\llbracket E \rrbracket_s : \mathsf{Cell}(u)]) & \rightsquigarrow_k (s, h) \\
\mathbf{assume}(G), (s, h) & \rightsquigarrow_k (s, h),\ \text{if } \llbracket G \rrbracket_s = \mathsf{true} \\
\mathbf{assume}(G), (s, h) & \not\rightsquigarrow_k \qquad\quad \text{if } \llbracket G \rrbracket_s = \mathsf{false} \\
\mathbf{init}_A(E), (s, h[\llbracket E \rrbracket_s : \mathsf{Cell}(u)]) & \rightsquigarrow_k (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, k, 1)]) \\
\mathbf{finalize}(E), (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, k, 1)]) & \rightsquigarrow_k (s, h[\llbracket E \rrbracket_s : \mathsf{Cell}(u)]) \\
\mathbf{acquire}(E), (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, 0, \pi)]) & \rightsquigarrow_k (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, k, \pi)]) \\
\mathbf{acquire}(E), (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, j, \pi)]) & \not\rightsquigarrow_k \qquad\quad \text{if } j > 0 \\
\mathbf{release}(E), (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, k, \pi)]) & \rightsquigarrow_k (s, h[\llbracket E \rrbracket_s : \mathsf{Lock}(A, 0, \pi)]) \\
C, (s, h) & \rightsquigarrow_k \top, \qquad \text{otherwise}
\end{array}$$

Fig. 8. Transition relation for atomic commands. $\not\rightsquigarrow_k$ is used to denote that the command does not fault, but gets stuck. $\top$ indicates that the command faults.

6 Interleaving Operational Semantics

Consider a program S consisting of a parallel composition of n threads. We abstract away from the particular syntax of the programming language and represent each thread by its control-flow graph (CFG). A CFG over a set $\mathcal{C}$ of atomic commands is defined as a tuple $(N, F, \mathsf{start}, \mathsf{end})$, where N is the set of program points, $F \subseteq N \times \mathcal{C} \times N$ the control-flow relation, start and end distinguished start and end program points. We note that a command in our language can be translated to a CFG. Conditional expressions in **if** and **while** commands are translated using the $\mathbf{assume}(G)$ statement that acts as a filter on the state space of programs – G is assumed to be true after $\mathbf{assume}(G)$ is executed. We let the set of atomic commands consist of primitive commands and the **assume** command. Let $(N_k, F_k, \mathsf{start}_k, \mathsf{end}_k)$ be the CFG of thread with identifier k and let $N = \bigcup_{k=1}^{n} N_k$ and $F = \bigcup_{k=1}^{n} F_k$. First, for each thread $k = 1..n$ and atomic command C we define a transition relation $\rightsquigarrow_k$ shown in Figure 8.

The interleaving operational semantics of the program S is defined by a transition relation $\rightarrow_S$ that transforms pairs of program counters (represented by mappings from thread identifiers to program points) $\mathsf{pc} \in \{1, \ldots, n\} \rightarrow N$ and states $\sigma \in \text{States} \cup \{\top\}$. The relation $\rightarrow_S$ is defined as the least one satisfying:

$$\frac{(v, C, v') \in F \quad k \in \{1, \ldots, n\} \quad C, (s, h) \rightsquigarrow_k \sigma}{\mathsf{pc}[k : v], (s, h) \rightarrow_S \mathsf{pc}[k : v'], \sigma}\ .$$

We denote with $\rightarrow_S^*$ the reflexive and transitive closure of $\rightarrow_S$. Let us denote with pc_0 the initial program counter $[1 : \mathsf{start}_1] \ldots [n : \mathsf{start}_n]$ and with pc_f the final one $[1 : \mathsf{end}_1] \ldots [n : \mathsf{end}_n]$. We say that the program S is *safe* when run from an initial state σ_0 if it is not the case that for some pc we have $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \top$.

```
$\{x, y \Vdash x \mapsto \_ * y \mapsto \_\}$
init$_{A,y}$(x);
init$_{B,x}$(y);
$\{x, y \Vdash A(x, y) * \mathsf{Locked}_A(x, y) * B(y, x) * \mathsf{Locked}_B(y, x)\}$
release(x);
$\{x, y \Vdash A(x, y) * \mathsf{Locked}_B(y, x)\}$
release(y);
$\{x, y \Vdash \mathsf{emp}_{\mathbf{h}}\}$
```

$$I_A(X, Y) \stackrel{\Delta}{=} \mathsf{emp}_{\mathbf{s}} \wedge B(Y, X) \quad \text{and} \quad I_B(X, Y) \stackrel{\Delta}{=} \mathsf{emp}_{\mathbf{s}} \wedge A(Y, X)$$

Fig. 9. A pathological situation

7 Soundness

As it stands now, the logic allows some unpleasant situations to happen: in certain cases the proof system may not be able to detect a memory leak. Figure 9 shows an example of this kind. We assume defined lock sorts A and B with invariants $I_A(X, Y)$ and $I_B(X, Y)$. In this case the knowledge that the locks at the addresses x and y exist is lost by the proof system: the invariant for the lock x holds the full permission for the handle of the lock y and vice versa, hence, local states of the threads are then left without any permissions for the locks whatsoever.

Situations such as the one described above make the formulation of the soundness statement for our logic non-trivial. We first formulate a soundness statement (Theorem 1) showing that every final state of a program (according to the operational semantics of Section 6) can be obtained as the $*$-conjunction of the postconditions of threads and the resource invariants *for the free locks allocated in the state.* Note that here a statement about a state uses the information about the free locks allocated in the same state. We then put restrictions on resource invariants that rule out situations similar to the one shown in Figure 9 and formulate a soundness statement (Theorem 4) in which the set of free locks in a final state is computed solely from the postconditions of threads.

For a state σ let $\mathsf{Free}(\sigma)$, respectively, $\mathsf{Unknown}(\sigma)$ be the set of pairs from $\mathcal{L} \times \mathrm{Locs}$ consisting of sorts and addresses of locks allocated in the state that have value 0, respectively, U. We denote with $\circledast$ iterated separating conjunction [15]: $\circledast_{j=1}^{k} P_j = (\mathsf{emp}_{\mathbf{s}} \wedge \mathsf{emp}_{\mathbf{h}}) * P_1 * \cdots * P_k$. The soundness of the logic with respect to the interleaving operational semantics from Section 6 is established by:

Theorem 1 *Let S be the program $C_1 \parallel \ldots \parallel C_n$ and suppose $\vdash \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$. Then for any interpretation i and state σ_0 such that $\sigma_0 \in \left(\circledast_{k=1}^{n} \llbracket P_k \rrbracket_i^k\right) * \left(\circledast_{(A,u) \in \mathsf{Free}(\sigma_0)} \mathcal{I}_0(A, u)\right)$ the program S is safe when run from σ_0 and if $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}_{\mathbf{f}}, \sigma$, then $\sigma \in \left(\circledast_{k=1}^{n} \llbracket Q_k \rrbracket_i^k\right) * \left(\circledast_{(A,u) \in \mathsf{Free}(\sigma)} \mathcal{I}_0(A, u)\right)$.*

The proof is given in Appendix B . We do not follow Brookes's original proof of soundness of concurrent separation logic [3]. Instead, we prove soundness with the aid of an intermediate thread-local semantics defined by fixed-point equations that can be viewed as the scheme of a thread-modular program analysis in the style of [7]. This method of proving soundness should facilitate designing program analyses based on our logic. The idea of our proof, however, is close to that of Brookes's and consists of establishing what is called the Separation Property in [11] and is formalized as the Parallel Decomposition Lemma in [3]:At any time, the state of the program can be partitioned into that owned by each thread and each free lock. As a direct consequence of the Separation Property, we can also show that provability of a program in our proof system ensures the absence of data races (see Appendix B.3 for details).

We now proceed to formulate a soundness statement in which the component $\circledast_{(A,u)\in\mathsf{Free}(\sigma)} \mathcal{I}_0(A,u)$ from Theorem 1 representing the resource invariants for free locks in the final state is obtained directly from the thread postconditions Q_k. To this end, we introduce an auxiliary notion of closure. Intuitively, closing a state amounts to $*$-conjoining it to the invariants of all free locks whose handles are reachable via resource invariants from the handles present in the state.

Definition 2 (Closure) *For $p \subseteq \mathrm{States}$ let $c(p) \subseteq \mathrm{States}$ be the least predicate such that $p \cup \{\sigma_1 * \sigma_2 \mid \sigma_1 \in c(p) \wedge \sigma_2 \in \circledast_{(A,u)\in\mathsf{Unknown}(\sigma_1)} \mathcal{I}_0(A,u)\} \subseteq c(p)$. The closure $\langle p\rangle$ of p is the set of states from $c(p)$ that do not contain locks with the value U.*

In general, the closure is not guaranteed to add invariants for all the free locks allocated in the state. For example, the closure of the postcondition of the program in Figure 9 still has an empty heap while in the final states obtained by executing the operational semantics there are locks allocated at addresses x and y. The problem is that there may exist a "self-contained" set of free locks (containing the locks at the addresses x and y in our example) such that the corresponding resource invariants hold full permissions for all the locks from the set. Local states of threads are then left without any permissions for the locks in the set, and hence, closure is not able to reach to their invariants. The following condition on resource invariants ensures that this does not happen.

Definition 3 (Admissibility of resource invariants) *Resource invariants for a set of lock sorts $\mathcal{L}$ are admissible if there do not exist non-empty set $L \subseteq \mathcal{L} \times \mathrm{Locs}$ and state $\sigma \in \circledast_{(A,u)\in L} \mathcal{I}_0(A,u)$ such that for all $(A,u) \in L$ the permission associated with the lock at the address u in σ is 1.*

Definitions 2 and 3 generalize to the case when resource invariants have more than one parameter in the obvious way. Revisiting Example 3 of Section 4, we can check that any state satisfying the closure of $[\![O \Vdash (1/M)H(\mathit{head})]\!]^k$ for any thread identifier k represents an acyclic sorted list starting at *head*. It is easy to check that resource invariants for the set of lock sorts $\{R,P,H,N\}$ from Section 4 are admissible whereas those for $\{A,B\}$ from this section are not. The admissibility of N is due to the fact that I_N implies sortedness of lists built out of resource invariants for N, hence, the invariants cannot form a cycle.

We say that a state is *complete* if permissions associated with all the locks allocated in it are equal to 1. Note that according to the semantics in Section 6, if σ_0 is complete and $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}, \sigma$, then σ is also complete. We can now formulate and prove the desired soundness statement.

Theorem 4 *Let S be the program $C_1 \parallel \ldots \parallel C_n$ and suppose $\vdash \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$. Suppose further that either at least one of Q_k is intuitionistic or resource invariants for lock sorts used in the proofs are admissible. Then for any interpretation i and complete state σ_0 such that $\sigma_0 \in \left\langle \circledast_{k=1}^{n} [\![P_k]\!]_i^k \right\rangle$ the program S is safe when run from σ_0 and if $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}_\mathbf{f}, \sigma$, then $\sigma \in \left\langle \circledast_{k=1}^{n} [\![Q_k]\!]_i^k \right\rangle$.*

Proof. Consider an interpretation i and a complete state $\sigma_0 \in \left\langle \circledast_{k=1}^{n} [\![P_k]\!]_i^k \right\rangle$. Therefore $\sigma_0 \in \left(\circledast_{k=1}^{n} [\![P_k]\!]_i^k\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(\sigma_0)} \mathcal{I}_0(A,u)\right)$ from the definition of closure. Then by Theorem 1 the program S is safe when run from σ_0 and if $\mathsf{pc}_0, \sigma_0 \rightarrow_S^* \mathsf{pc}_\mathbf{f}, \sigma$, then $\sigma \in \left(\circledast_{k=1}^{n} [\![Q_k]\!]_i^k\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(\sigma)} \mathcal{I}_0(A,u)\right)$. Hence, by the definition of closure, we have $\sigma \in \sigma_1 * \sigma_2$ where $\sigma_1 \in \left\langle \circledast_{k=1}^{n} [\![Q_k]\!]_i^k \right\rangle$ and $\sigma_2 \in \circledast_{(A,u)\in L} \mathcal{I}_0(A,u)$ for some $L \subseteq \mathsf{Free}(\sigma)$. If one of Q_k is intuitionistic, then from this it directly follows that $\sigma \in \left\langle \circledast_{k=1}^{n} [\![Q_k]\!]_i^k \right\rangle$.

Suppose now that $L \neq \emptyset$ and the resource invariants for lock sorts mentioned in L are admissible. Consider any $(A,u) \in L$. The state σ is complete, therefore, the permission associated with the lock at the address u in σ is 1. Besides, since $L \subseteq \mathsf{Free}(\sigma)$, the value associated with u in σ is 0. Hence, if the permission associated with u in σ_2 were less than 1, then u would have to be allocated in σ_1 with a non-zero permission and the value U, which would contradict the definition of closure (a state in a closure cannot contain locks with the value U). So, for any $(A,u) \in L$ the permission associated with u in σ_1 is 1, which contradicts the admissibility of resource invariants for lock sorts used in the proof of the program. Therefore, $L = \emptyset$ and, hence, $\sigma \in \left\langle \circledast_{k=1}^{n} [\![Q_k]\!]_i^k \right\rangle$. □

Note that for garbage-collected languages we can use the intuitionistic version of the logic [9] (i.e., one in which every assertion is intuitionistic) and, hence, do not have to check admissibility. Also, admissibility does not have to be checked if we are not interested in detecting memory leaks, as then Theorem 1 can be used.

8 Dynamic Thread Creation

We now extend the programming language with dynamically created threads:

$T ::= f, f_1, f_2, \ldots$	procedure names
$C ::= \ldots \mid V = \mathbf{fork}(T) \mid \mathbf{join}(E)$	primitive commands
$P ::= \mathbf{let}\ T = S,\ \ldots,\ T = S\ \mathbf{in}\ S$	programs

We represent the code of threads by parameterless procedures (passing parameters to threads at the time of their creation is orthogonal to our concerns here and can be handled in a way similar to the one used for handling procedure calls when variables are treated as resources [14]; see Appendix C.4 for details). A program consists of several procedure declarations along with the code of the main thread. We consider only well-formed programs in which all declared procedure names are distinct and all procedures used are declared. $x = \mathbf{fork}(f)$ creates a new thread executing the code of the procedure f and stores the corresponding thread identifier into the variable x. $\mathbf{join}(E)$ waits until the thread with the identifier E finishes executing. In our semantics we allow at most one thread to wait for the termination of a given thread.

We add two new forms to our assertion language: $\Phi ::= \ldots \mid \mathsf{tid}_f(E) \mid \mathsf{emp_t}$. A formula $\mathsf{tid}_f(E)$, which we call a thread handle, represents the knowledge that the thread with the identifier E exists and executes the code of the procedure f, and gives its owner a permission to join the thread. $\mathsf{emp_t}$ denotes that the assertion does not contain any permissions of this kind. Note that a thread is deallocated (only) when it is joined.

Judgements are now of the form $\Gamma \vdash \{P\}\ C\ \{Q\}$ where Γ is a context consisting of a set of procedure specifications, each of the form $\{P\}\ f\ \{Q\}$. We consider only contexts in which there is at most one specification for each procedure. As procedures are parameterless, we restrict our attention here to contexts in which pre- and postconditions do not contain free logical variables. We add $\Gamma\ \vdash$ to all the triples in the rules from Figure 1 as well as in the standard rules of separation logicfrom Appendix A. In addition, we $\wedge$-conjoin $\mathsf{emp_t}$ to every pre- and postcondition in the axioms for primitive commands (except for the postcondition of Acquire and the precondition of Release: in the postcondition of Acquire and the precondition of Release we $\wedge$-conjoin $\mathsf{emp_t}$ right after $\mathsf{Locked}_A(L, \vec{X})$). To reason about **fork** and **join** we introduce two new axioms:

$$\frac{}{\Gamma,\ \{P\}\ f\ \{Q\} \vdash \{(x \Vdash \mathsf{emp_h} \wedge \mathsf{emp_t}) * P\}\ x = \mathbf{fork}(f)\ \{x \Vdash \mathsf{emp_h} \wedge \mathsf{tid}_f(x)\}}\ \textsc{Fork}$$

$$\frac{}{\Gamma,\ \{P\}\ f\ \{Q\} \vdash \{O \Vdash \mathsf{emp_h} \wedge \mathsf{tid}_f(E)\}\ \mathbf{join}(E)\ \{(O \Vdash \mathsf{emp_h} \wedge \mathsf{emp_t}) * Q\}}\ \textsc{Join}$$

That is, upon creating a new thread executing the code of procedure f, the thread that executed **fork** obtains the thread handle $\mathsf{tid}_f(x)$ for the newly-created thread and gives up ownership of the precondition of f. Joining a thread with the identifier E requires the joining thread to own the handle $\mathsf{tid}_f(E)$. When **join** succeeds, the thread exchanges the handle for the postcondition of f.

The model of the assertion language has to be adapted to account for thread handles. A state of the program is now represented by a triple of a stack, a heap, and a thread pool, the latter represented by a finite partial function from thread identifiers to procedure names. Now a lock can be free, held by the main thread, held by a thread, or its status may be unknown. Assertions are then interpreted with respect to a thread identifier, a stack, a heap, a thread pool, and an interpretation of logical variables. We define the $*$ operation on thread pools as disjoint union of the partial functions representing them, and the semantics of $P * Q$ and $P \mathrel{-\!\!*} Q$ are then straightforwardly adjusted so as to partition thread pools. In addition, we add clauses for the new forms in our assertion language such that $\mathsf{tid}_f(E)$ describes singleton thread pools and $\mathsf{emp_t}$ describes the empty thread pool. The satisfaction relation for all other formulae just ignores the thread pool. The notion of precision of formulae does not change.

In this section, we omit the detailed development of semantics and soundness for the extended logic, which can be found in Appendix C . Instead, we just state the conditions under which the logic is sound. A proof of the program

let $f_1 = C_1,\ \ldots,\ f_n = C_n$ **in** C is given by triples $\Gamma \vdash \{P_1\}\ C_1\ \{Q_1\}, \ldots, \Gamma \vdash \{P_n\}\ C_n\ \{Q_n\}, \Gamma \vdash \{P\}\ C\ \{Q\}$, where $\Gamma = \{P_1\}\ f_1\ \{Q_1\},\ \ldots,\ \{P_n\}\ f_n\ \{Q_n\}$. For the proof to be sound, P_k must be precise, and P_k and Q_k must have empty locksets, for all $k = 1..n$. We note that the operational semantics of Section 6 can be adjusted to our setting and a soundness statement similar to Theorem 1 can then be formulated. The proof of soundness is then done in the same style as that of Theorem 1. The notions of closure and admissibility can also be generalized to the new setting and a theorem similar to Theorem 4 can be proved.

9 Conclusions and Related Work

We have presented a logic that allows reasoning about concurrent heap-manipulating programs with realistic concurrency primitives including unbounded numbers of locks dynamically allocated and destroyed in the heap and threads dynamically created and terminating themselves. We have demonstrated that the logic makes it possible to reason locally about programs with a notion of dynamic ownership of heap parts by locks and threads. We believe that in the future this aspect of the logic will produce some additional pay-offs. First, the resource-oriented flavor of the logic should make it easy to design program analyses on the basis of it following the lines of [7]. In fact, the fixed-point equations defining the thread-local semantics used in the proof of soundness of our logic (Appendix B) can be seen as a scheme of a thread-modular program analysis in the style of [7]. Second, lock handles in our logic's assertion language are somewhat reminiscent of abstract predicates used for modular reasoning in separation logic about object-oriented programs [13]. This is not a coincidence as object-oriented programs use information hiding extensively in their locking mechanisms, and hence, often satisfy the Ownership Hypothesis. For this reason, we believe that our logic, combined with the techniques from [13], should be convenient for reasoning about concurrent object-oriented programs. Note, however, that lock handles and abstract predicates are different, in particular, we cannot see a way in which the former can be encoded in terms of the latter.

Two papers [5, 17] have recently suggested combinations of separation logic and rely-guarantee reasoning that, among other things, can be used to reason about storable locks. For example, in [17] locks are not treated natively in the logic, but are represented as cells in memory storing the identifier of the thread that holds the lock; rely-guarantee is then used to simplify reasoning about the global shared heap with locks allocated in it. The logic allows modular reasoning about complex fine-grained concurrency algorithms (e.g., about the optimistic list mentioned in Section 4), but loses locality of reasoning for programs that allocate and deallocate many simple data structures protected by locks, which results in awkward proofs. In other words, as the original concurrent separation logic, the logics in [5, 17] are designed for reasoning about the concurrent control of bounded numbers of data structures whereas our logic is designed to reason about the concurrent control of unboundedly many data structures that are dynamically created and destroyed. Ideally, one wants to have a combination of both: a logic in which on the higher-level the reasoning is performed in a resource-oriented fashion and on the lower-level rely-guarantee is applied to deal with complex cases. Achieving this is another direction of our future research.

Feng and Shao [6] presented a rely-guarantee logic for reasoning about concurrent assembly code with dynamic thread creation. They do not have analogs of our rules for ownership transfer at **fork** and **join** commands. On a higher level, our logic for storable threads relates to theirs in the same way as separation logic relates to rely-guarantee reasoning: the former is good at describing ownership transfer, the latter at describing interference. As in the case of storable locks, investigating possible combinations of the two approaches would be fruitful.

Acknowledgments. We would like to thank Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson for comments and discussions that helped to improve the paper.

References

1. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
2. S. D. Brookes. Variables as resource for shared-memory programs: Semantics and soundness. In *MFPS*, 2006.
3. S. D. Brookes. A semantics of concurrent separation logic. *Theoretical Computer Science*, 375(1-3):227–270, 2007. Preliminary version appeared in *CONCUR*, 2004.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.
5. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.
6. X. Feng and Z. Shao. Modular verification of concurrent assembly code with dynamic thread creation and termination. In *ICFP*, 2005.
7. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI*, 2007.
8. J. Hayman and G. Winskel. Independence and concurrent separation logic. In *LICS*, 2006.
9. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
10. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, 2001.
11. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. Preliminary version appeared in *CONCUR*, 2004.
12. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, 2004.
13. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, 2005.
14. M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logics. In *LICS*, 2006.
15. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
16. V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP*, 2006.
17. V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

A Proof Rules of Separation Logic

$$\frac{}{\{x, O \Vdash X = E \wedge \mathsf{emp}_{\mathbf{h}}\}\ x = E\ \{x, O \Vdash x = X \wedge \mathsf{emp}_{\mathbf{h}}\}}\ \textsc{Assn}$$

$$\frac{(O \Vdash E \mapsto \_) \Rightarrow F = F}{\{O \Vdash E \mapsto \_\}\ [E] = F\ \{O \Vdash E \mapsto F\}}\ \textsc{Mutate}$$

$$\frac{}{\{x, O \Vdash X = E \wedge X \mapsto Y\}\ x = [E]\ \{x, O \Vdash x = Y \wedge X \mapsto Y\}}\ \textsc{Lookup}$$

$$\frac{}{\{x \Vdash \mathsf{emp}_{\mathbf{h}}\}\ x = \mathbf{new}\ \{x \Vdash x \mapsto \_\}}\ \textsc{New}$$

$$\frac{}{\{O \Vdash E \mapsto \_\}\ \mathbf{delete}\ E\ \{O \Vdash \mathsf{emp}_{\mathbf{h}}\}}\ \textsc{Delete}$$

$$\frac{\{P_1\}\ C\ \{Q_1\} \quad \{P_2\}\ C\ \{Q_2\}}{\{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}\ \textsc{Conj} \qquad \frac{\{P_1\}\ C\ \{Q_1\} \quad \{P_2\}\ C\ \{Q_2\}}{\{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}\ \textsc{Disj}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{\exists X.P\}\ C\ \{\exists X.Q\}}\ \textsc{Exists} \qquad \frac{P_1 \Rightarrow P_2 \quad \{P_2\}\ C\ \{Q_2\} \quad Q_2 \Rightarrow Q_1}{\{P_1\}\ C\ \{Q_1\}}\ \textsc{Conseq}$$

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \textsc{Frame} \qquad \frac{\{P\}\ C_1\ \{Q\} \quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}\ \textsc{Seq}$$

$$\frac{P \Rightarrow B = B \quad \{P \wedge B\}\ C_1\ \{Q\} \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ \{Q\}}\ \textsc{Cond}$$

$$\frac{P \Rightarrow B = B \quad \{P \wedge B\}\ C\ \{P\}}{\{P\}\ \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od}\ \{P \wedge \neg B\}}\ \textsc{While}$$

B Proof of Soundness

So far most of the proofs of soundness of concurrent separation logic with respect to an independently stated semantics [3, 4, 8, 2] have followed Brookes's original proof [3] based on trace semantics. Here we take a different approach. To prove the soundness of our logic, we first define a thread-local forward predicate transformer semantics and define the notion of validity of Hoare triples for commands with respect to this semantics (recall that commands in our terminology do not contain parallel composition). We prove the soundness of all the proof rules with respect to the thread-local semantics (Section B.1). We then prove that the thread-local semantics is in some sense adequate with respect to the interleaving operational semantics (Lemma 11), which lets us justify the soundness of our logic (Section B.2). In addition, using the semantics we prove that the provability of a program in our logic implies that the program is data race free (Section B.3).

The approach described above has two major benefits. First, the proof of soundness done according to it is very simple in comparison with the prior ones, both conceptually and technically. Second, the fixed-point equations defining the thread-local semantics in Section B.1 can be viewed as the scheme of a thread-modular program analysis in the style of [7] and the over-approximation lemma in Section B.2 as the statement of its soundness. Hence, our semantics dictates the design of an automatic program analysis based on the logic (implementing such an analysis is a subject of our current work).

B.1 Thread-local semantics

Let the domain $\mathcal{D}$ be the topped powerset of states defined in Figure 6 with the order $\sqsubseteq$ (the subset inclusion with $\top$ being the topmost element), join $\sqcup$ and meet $\sqcap$ operators. It is sometimes convenient to use an alternate representation in terms of the isomorphic domain $\mathcal{P}(\text{States} \cup \{\top\})$. The isomorphism between $\mathcal{P}(\text{States} \cup \{\top\})$ and $\mathcal{D}$ associates any subset of $\text{States} \cup \{\top\}$ that contains $\top$ with $\top$ in $\mathcal{D}$, and any other subset with the corresponding element of $\mathcal{D}$. We lift $*$ to operate on $\mathcal{D}$ letting $p * \top = \top * p = \top$ for all p. We let $\mathsf{Free}(\top) = \emptyset$.

We now define a thread-local forward predicate transformer $\mathsf{Post}_k(C) : \mathcal{D} \rightarrow \mathcal{D}$ for every thread k and command C. We first define Post_k for atomic commands C other than **acquire** and **release** using the transition relation $\rightsquigarrow_k$ from Figure 8:

$$\mathsf{Post}_k(C, (s, h)) = \{q \mid C, (s, h) \rightsquigarrow_k q\}.$$

We now let

$$\mathsf{Post}_k(\mathbf{acquire}(E), (s, h)) = \{(s, h) * r \mid r \in \mathcal{I}_k(A, [\![E]\!]_s)\},$$

if $[\![E]\!]_s\!\downarrow$ and $h([\![E]\!]_s) = \mathsf{Lock}(A, v, \pi)$, otherwise $\mathsf{Post}_k(\mathbf{acquire}(E), (s, h)) = \top$. For a state q and a precise predicate p let

$$\mathsf{rest}(q, p) = \begin{cases} q_1, & \text{if } q = q_1 * q_2 \text{ and } q_2 \in p; \\ \top, & \text{otherwise.} \end{cases}$$

rest is well-defined because p is precise. We then let

$$\mathsf{Post}_k(\mathbf{release}(E), (s, h)) = \{\mathsf{rest}((s, h), \mathcal{I}_k(A, [\![E]\!]_s))\},$$

if $[\![E]\!]_s\!\downarrow$ and $h([\![E]\!]_s) = \mathsf{Lock}(A, v, \pi)$, otherwise $\mathsf{Post}_k(\mathbf{release}(E), (s, h)) = \top$. We lift Post to $\mathcal{D}$ pointwise and let $\mathsf{Post}_k(C, \top) = \top$ for all atomic C.

Note that predicate transformers mimic the proof rules for commands: in particular, upon acquiring a lock the thread gets the ownership of the resource invariant associated with the lock; upon releasing a lock the thread gives up the ownership of the invariant. (Recall that $\mathcal{I}_k(A, u)$ includes the interpretation of the Locked fact associated with the lock at the address u.) In the latter case the requirement that $\mathcal{I}_k(A, u)$ be precise for each k and u ensures the determinacy of splitting the state upon releasing a lock. The annotations of lock sorts and parameters at **init** commands ensure that the predicate transformer for each **init** command in a program is defined uniquely.

In order to define $\mathsf{Post}_k(C, p)$ for a composite command C we first translate it to a CFG $(N, F, \mathsf{start}, \mathsf{end})$ assuming, without loss of generality, that there are no edges in F leading to start or going out of end. We then condiser a functional $F_k(C, p) : (N \to \mathcal{D}) \to (N \to \mathcal{D})$ defined in the following way: $F_k(C, p)(g) = g'$ where $g'(\mathsf{start}) = p$ and for every node $v_2 \in N$ such that $v_2 \neq \mathsf{start}$, $g'(v_2) = \bigsqcup_{(v_1,C,v_2)\in F} \mathsf{Post}_k(C, g(v_1))$. That is, to compute the set of reachable states at any node (except for the initial one) we consider all the edges in the CFG that can be taken by C that lead to this node and take the join of predicate transformers for the commands at these edges with respect to the states at source nodes. It is easy to see that $\mathsf{Post}_k(C)$ for atomic commands are monotone, hence, by Tarski's fixed-point theorem the functional has least (under the pointwise extension of $\sqsubseteq$) fixed point $\mathsf{lfp}(F_k(C, p))$. Let $\mathsf{Post}_k(C, p) = (\mathsf{lfp}(F_k(C, p)))(\mathsf{end})$. We can now define the notion of validity $\models_k$ of Hoare triples with respect to the thread-local semantics for thread k.

Definition 5 *For a command* $C \;\; \models_k \{P\}\ C\ \{Q\} \Leftrightarrow \forall i. \mathsf{Post}_k(C, [\![P]\!]^k_i) \sqsubseteq [\![Q]\!]^k_i$.

We say that an inference rule is sound with respect to the thread-local semantics if for all $k \in$ ThreadIDs whenever all of its premisses are semantically valid (as defined by the relation $\models_k$ above), the conclusion is also semantically valid. We now proceed to prove that all our proof rules are sound with respect to the thread-local semantics.

Lemma 6 *Rules* Assn*,* Mutate*,* Lookup*,* New*,* Delete*,* Exists*,* Conseq*,* Seq*,* Cond*, and* While *are sound with respect to the thread-local semantics.*

Proof. An adaptation of proofs from [9, 14] to our setting. □

Lemma 7 *Rules* Init*,* Finalize*,* Acquire*, and* Release *are sound with respect to the thread-local semantics.*

Proof. Follows easily from the definition of Post_k for these commands. □

Lemma 8 *For all* $k \in$ ThreadIDs*,* $p, q \in \mathcal{D}$ *and commands* C

1. $\mathsf{Post}_k(C, p \sqcap q) \sqsubseteq \mathsf{Post}_k(C, p) \sqcap \mathsf{Post}_k(C, q)$.
2. $\mathsf{Post}_k(C, p \sqcup q) \sqsubseteq \mathsf{Post}_k(C, p) \sqcup \mathsf{Post}_k(C, q)$.
3. $\mathsf{Post}_k(C, p * q) \sqsubseteq \mathsf{Post}_k(C, p) * q$.

Proof. In the case of $\sqcap$ the required follows from the monotonicity of $\mathsf{Post}_k(C)$. For the other two cases the inclusions are first established for atomic commands using the definition of Post_k. The case of composite commands is then discharged via Park induction. □

Corollary 9 *Rules* Conj*,* Disj*, and* Frame *are sound with respect to the thread-local semantics.*

Lemma 10 (Soundness with respect to the thread-local semantics) *If* $\vdash \{P\}\ C\ \{Q\}$*, then for all* $k \in \text{ThreadIDs}$ $\models_k \{P\}\ C\ \{Q\}$*.*

Proof. Induction on the derivation of $\{P\}\ C\ \{Q\}$ using Lemmas 6 and 7 and Corollary 9. □

B.2 The proof

We first show that the thread-local semantics in some sense over-approximates the interleaving operational semantics of Section 6. This is formally stated in the following lemma.

Lemma 11 (Over-approximation Lemma) *Let the program* S *be* $C_1 \parallel \ldots \parallel C_n$*,* $q_0 \in \text{States}$ *and* $p_1, \ldots, p_n \in \mathcal{D}$ *be such that*

$$\{q_0\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} p_k \right) * \left(\mathop{\circledast}_{(A,u)\in \mathsf{Free}(q_0)} \mathcal{I}_0(A,u) \right), \tag{1}$$

and $g_k = \mathsf{lfp}(F_k(C_k, p_k))$ *for* $k = 1..n$*. Then whenever* $\mathsf{pc}_0, q_0 \to_S^* \mathsf{pc}, q$*, it is the case that*

$$\{q\} \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} g_k(\mathsf{pc}(k)) \right) * \left(\mathop{\circledast}_{(A,u)\in \mathsf{Free}(q)} \mathcal{I}_0(A,u) \right). \tag{2}$$

Before proving the lemma we introduce an auxiliary notion of transfer functions. For a state (s,h) and an atomic command C the *transfer function* $f_C^k(s,h)$ computes all the states to which (s,h) can be transformed when C is executed by thread k: $f_C^k(s,h) = \{q \mid C,(s,h) \rightsquigarrow_k q\}$. We let $f_C^k(\top) = \top$ and lift f_C^k to $\mathcal{D}$ pointwise. Note that f_C^k are monotone and for commands other than **acquire** and **release** $f_C^k(q) = \mathsf{Post}_k(C,q)$. A fundamental property of the semantics of Section 6 is that the transfer functions built according to it are *local* in the following sense.

Lemma 12 (Locality Lemma) *For all* $p, q \in \mathcal{D}$ *and atomic commands* C $f_C^k(p * q) \sqsubseteq f_C^k(p) * q$*.*

Proof of Lemma 11. We prove the statement of the theorem by induction on the length m of the derivation of q. For $m = 0$ (2) is equivalent to (1). Suppose now that $\mathsf{pc}_0, q_0 \to_S^* \mathsf{pc}[j : v], q$, (2) is fulfilled, and $\mathsf{pc}[j : v], q \to_S \mathsf{pc}[j : v'], q'$. Then $(v, C, v') \in F$, $C, q \rightsquigarrow_j q'$, and, hence, $\{q'\} \sqsubseteq f_C^j(q)$. We then have to show that

$$f_C^j(\{q\}) \sqsubseteq \left(\mathop{\circledast}_{k=1}^{n} g_k(\mathsf{pc}(k)) \right) * \left(\mathop{\circledast}_{(A,u)\in \mathsf{Free}(q)} \mathcal{I}_0(A,u) \right). \tag{3}$$

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

1. C is not **acquire** *or* **release**. Let $q_1 = g_j(v)$, $q_2 = g_j(v')$, and

$$r = \left(\circledast_{\substack{1 \leq k \leq m, \\ k \neq j}} g_k(\mathsf{pc}(k)) \right) * \left(\circledast_{(A,u) \in L} \mathcal{I}_0(A,u) \right), \tag{4}$$

where $L = \mathsf{Free}(q)$. Then $\{q\} \sqsubseteq q_1 * r$. We have to prove that $f_C^j(q) \sqsubseteq q_2 * r$. For this it is sufficient to prove that $f_C^j(q_1 * r) \sqsubseteq q_2 * r$. From the definition of the functional $F_j(C_j, p_j)$ we get $f_C^j(q_1) = \mathsf{Post}_j(C, q_1) \sqsubseteq q_2$. From this and Lemma 12 we get $f_C^j(q_1 * r) \sqsubseteq f_C^j(q_1) * r \sqsubseteq q_2 * r$.

2. C is **acquire**(E). We can assume that $\{q\} \sqsubseteq \{q_1\} * \mathcal{I}_0(A,u) * r$, where $\{q_1\} \sqsubseteq g_j(v)$, $q_1 = (s, h[u : \mathsf{Lock}(A, \mathsf{U}, \pi)])$, $[\![E]\!]_s = u$, $(A,u) \in \mathsf{Free}(q)$, and r is defined by (4) with $L = \mathsf{Free}(q) \backslash \{(A,u)\}$; otherwise either the right-hand side of (3) is $\top$ or its left-hand side is $\emptyset$. Let $q_2 = g_j(v')$. We have to show that $f_C^j(q) \sqsubseteq q_2 * r$. For this it is sufficient to show that $f_C^j(\{q_1\} * \mathcal{I}_0(A,u) * r) \sqsubseteq q_2 * r$. From the definitions of the functional $F_j(C_j, p_j)$ and Post_j for **acquire** we get

$$\{q_1\} * \mathcal{I}_j(A,u) \sqsubseteq q_2. \tag{5}$$

Hence,

$$\begin{aligned} f_C^j(\{q_1\} * \mathcal{I}_0(A,u) * r) &\sqsubseteq f_C^j(\{q_1\} * \mathcal{I}_0(A,u)) * r && \text{Lemma 12} \\ &= \{q_1\} * \mathcal{I}_j(A,u) * r && I_A(L) \text{ has an empty lockset} \\ &\sqsubseteq q_2 * r && \text{by (5)} \end{aligned}$$

3. C is **release**(E). We can assume that $\{q\} \sqsubseteq q_1 * r$, where $\{q_1\} \sqsubseteq g_j(v)$, $q_1 = (s, h[u : \mathsf{Lock}(A, j, \pi)])$, $[\![E]\!]_s = u$, $(A,u) \notin \mathsf{Free}(q)$, and r is defined by (4) with $L = \mathsf{Free}(q)$; otherwise the right-hand side of (3) is $\top$. We have to prove that $f_C^j(q) \sqsubseteq q_2 * \mathcal{I}_0(A,u) * r$. For this it is sufficient to show that $f_C^j(\{q_1\} * r) \sqsubseteq q_2 * \mathcal{I}_0(A,u) * r$. From the definitions of the functional $F_j(C_j, p_j)$ and Post_j for **release** we get $\{\mathsf{rest}(q_1, \mathcal{I}_j(A,u))\} \sqsubseteq q_2$. Then using the definition of rest we obtain

$$\{q_1\} \sqsubseteq \{\mathsf{rest}(q_1, \mathcal{I}_j(A,u))\} * \mathcal{I}_j(A,u) \sqsubseteq q_2 * \mathcal{I}_j(A,u). \tag{6}$$

Hence,

$$\begin{aligned} f_C^j(\{q_1\} * r) &\sqsubseteq f_C^j(q_1) * r && \text{Lemma 12} \\ &\sqsubseteq f_C^j(q_2 * \mathcal{I}_j(A,u)) * r && \text{by (6)} \\ &= q_2 * \mathcal{I}_0(A,u) * r && I_A(L) \text{ has an empty lockset} \end{aligned}$$

So, in all cases (3) is fulfilled, which implies the statement of the theorem. □

We are now in a position to prove the main soundness theorem.

Proof of Theorem 1. Consider an interpretation i and a state q_0 such that $q_0 \in \left(\circledast_{k=1}^{n} [\![P_k]\!]_i^k\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(q_0)} \mathcal{I}_0(A,u)\right)$. Let $p_k = [\![P_k]\!]_i^k$ in Lemma 11, then (1) is fulfilled. By Lemma 10, $\vDash_k \{P_k\}\ C_k\ \{Q_k\}$ for $k = 1..n$, hence, by Definition 5, $\mathsf{Post}_k(C_k, [\![P_k]\!]_i^k) \sqsubseteq [\![Q_k]\!]_i^k \sqsubset \top$. In particular, for any pc and q such that $\mathsf{pc}_0, q_0 \to_S^* \mathsf{pc}, q$, from (2) it then follows that $\{q\} \sqsubset \top$, i.e., S is safe when run from q_0. Now letting $\mathsf{pc} = \mathsf{pc_f}$ and using (2) we get $q \in \left(\circledast_{k=1}^{n} [\![Q_k]\!]_i^k\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(q)} \mathcal{I}_0(A,u)\right)$. □

B.3 Data race freedom

We show that the provability of a program in our logic implies that the program is data race free. We assume that the program is represented by a CFG and use the notation defined in Section 6 and Lemma 11.

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

Definition 13 (Interfering commands) *Atomic commands C' and C'' interfere with each other when executed from the state (s, h), denoted with $C' \bowtie_{(s,h)} C''$, if $\mathsf{accesses}(C', s, h) \cap \mathsf{writes}(C'', s, h) \neq \emptyset$ or $\mathsf{writes}(C', s, h) \cap \mathsf{accesses}(C'', s, h) \neq \emptyset$.*

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

Definition 14 (Data race) *The program S has a data race when run from an initial state q_0 if for some pc and state (s, h) such that $\mathsf{pc}_0, q_0 \to_S^* \mathsf{pc}, (s, h)$, there exist CFG edges $(v_1, C', v_1') \in F_j$ and $(v_2, C'', v_2') \in F_k$ $(j \neq k)$ labeled with atomic commands C' and C'' such that $C', (s, h) \not\rightsquigarrow_j \top$, $C'', (s, h) \not\rightsquigarrow_k \top$, and $C' \bowtie_{(s,h)} C''$.*

Theorem 15 (Data race freedom) *Let S be the program $C_1 \parallel \ldots \parallel C_n$ and suppose that the triples $\{P_k\}\ C_k\ \{Q_k\}$ $(k = 1..n)$ are provable in our logic. Then the program S has no data races when run from initial states q_0 such that $q_0 \in \left(\circledast_{k=1}^{n} [\![P_k]\!]_i^k\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(q_0)} \mathcal{I}_0(A,u)\right)$ for any interpretation i.*

Proof. Suppose the contrary: there exist an interpretation i and an initial state q_0 such that $q_0 \in \left(\circledast_{k=1}^{n} [\![P_k]\!]_i^k\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(q_0)} \mathcal{I}_0(A,u)\right)$, a program counter pc and a state (s, h) such that $\mathsf{pc}_0, q_0 \to_S^* \mathsf{pc}, (s, h)$, CFG edges $(v_1, C', v_1') \in F_j$ and $(v_2, C'', v_2') \in F_k$ $(j \neq k)$ labeled with atomic commands C' and C'' such that $C', (s, h) \not\rightsquigarrow_j \top$, $C'', (s, h) \not\rightsquigarrow_k \top$, and $C' \bowtie_{(s,h)} C''$.

Let $q_1 = g_j(v_j)$, $q_2 = g_k(v_k)$, then by Lemma 11 for some r, $(s, h) \in r * q_1 * q_2$. Hence,

$$(s, h) = (s_0, h_0) * (s_1, h_1) * (s_2, h_2), \tag{7}$$

where

$$(s_0, h_0) \in r, \quad (s_1, h_1) \in q_1, \quad (s_2, h_2) \in q_2. \tag{8}$$

Since $\vdash \{P_j\}\ C_j\ \{Q_j\}$ and $\vdash \{P_k\}\ C_k\ \{Q_k\}$, from Definition 5 it follows that $\mathsf{Post}_j(C', q_1) \sqsubset \top$ and $\mathsf{Post}_k(C'', q_2) \sqsubset \top$. From this and (8) we obtain $f^j_{C'}(s_1, h_1) \sqsubseteq f^j_{C'}(q_1) = \mathsf{Post}_j(C', q_1) \sqsubset \top$. So, $f^j_{C'}(s_1, h_1) \sqsubset \top$ and, analogously, $f^k_{C''}(s_2, h_2) \sqsubset \top$. Hence, $C', (s_1, h_1) \not\rightarrowtail_j \top$ and $C'', (s_2, h_2) \not\rightarrowtail_k \top$. From this and the fact that $C' \bowtie_{(s,h)} C''$ using the definition of $*$ and the transition relation for atomic commands given in Section 6 we easily get that $(s_1, h_1) * (s_2, h_2)$ is undefined, which contradicts (7). The intuition behind this is that from $C', (s_1, h_1) \not\rightarrowtail_j \top$ and $C'', (s_2, h_2) \not\rightarrowtail_k \top$ it follows that both (s_1, h_1) and (s_2, h_2) should have the full permission for the same variable or location accessed by C' and C'', which makes the state $(s_1, h_1) * (s_2, h_2)$ inconsistent. □

C More on Dynamic Thread Creation

We first give some formal details on the model of the extended assertion language introduced in Section 8. The model of the assertion language from Figure 6 is changed to account for thread handles in the following way:

$$\begin{array}{ll}
\mathrm{LockVals} = \{\mathsf{U}, \mathsf{M}, 0\} \cup \mathrm{ThreadIDs} & \mathrm{ThreadPools} = \mathrm{ThreadIDs} \rightharpoonup_{\mathrm{fin}} \mathrm{Procs} \\
\mathrm{Procs} = \{f, f_1, f_2, \dots\} & \mathrm{States} = \mathrm{Stacks} \times \mathrm{Heaps} \times \mathrm{ThreadPools}
\end{array}$$

Lock values are interpreted as follows: 0 represents a free lock, an identifier from ThreadIDs signifies that the lock is held by the thread with this identifier, M means that the lock is held by the main thread, and U shows that the status of the lock is unknown.

An assertion is interpreted with respect to a thread identifier $k \in \{\mathsf{M}, 0\} \cup$ ThreadIDs, a stack s, a heap h, a thread pool t, and an interpretation of logical variables i: $(s, h, t, i) \models_k \Phi$. The satisfaction relation for the new assertion forms is as follows:

$$\begin{array}{ll}
(s, h, t, i) \models_k \mathsf{tid}_f(E) & \Leftrightarrow [\![E]\!]_{(s,i)}\!\downarrow \wedge\, t = [[\![E]\!]_{(s,i)} : f] \\
(s, h, t, i) \models_k \mathsf{emp_t} & \Leftrightarrow t = [\,]
\end{array}$$

C.1 Operational semantics and soundness

Let S be the program **let** $f_1 = C_1,\ \dots,\ f_n = C_n$ **in** C. As before, we assume that C_k, respectively, C are represented by CFGs $(N_{f_k}, F_{f_k}, \mathsf{start}_{f_k}, \mathsf{end}_{f_k})$, respectively, $(N_{\mathsf{main}}, F_{\mathsf{main}}, \mathsf{start}_{\mathsf{main}}, \mathsf{end}_{\mathsf{main}})$, and let $N = \bigcup_{k=1}^{n} N_{f_k} \cup N_{\mathsf{main}}$ and $F = \bigcup_{k=1}^{n} F_{f_k} \cup F_{\mathsf{main}}$. Let $\mathsf{proc}(v)$ denote the name of the procedure to which the program point v belongs (main for program points in the main thread).

The interleaving operational semantics of the program S is defined by a transition relation $\rightarrow_S$ that transforms pairs of program counters $\mathsf{pc} \in (\mathrm{ThreadIDs} \cup \{\mathsf{M}\}) \rightharpoonup_{\mathrm{fin}} N$ and states (s, h, t) such that $\mathrm{dom}(t) \cup \{\mathsf{M}\} = \mathrm{dom}(\mathsf{pc})$ and $\forall k. t(k)\!\downarrow\ \Rightarrow\ \mathsf{proc}(\mathsf{pc}(k)) = t(k)$. The relation $\rightarrow_S$ is defined as the least one satisfying the rules in Figure 10. We denote with pc_0 the initial program counter $[\mathsf{M} : \mathsf{start}_{\mathsf{main}}]$. The notion of program safety stays the same as before. We redefine iterated separating conjunction as $\circledast_{j=1}^{k} P_j = (\mathsf{emp_s} \wedge \mathsf{emp_h} \wedge \mathsf{emp_t}) * P_1 * \cdots * P_k$.

$$\frac{C,(s,h) \rightsquigarrow_k (s',h')}{C,(s,h,t) \rightsquigarrow_k (s',h',t)}, \quad \frac{C,(s,h) \rightsquigarrow_k \top}{C,(s,h,t) \rightsquigarrow_k \top} \quad (C \text{ is not } \textbf{fork} \text{ or } \textbf{join})$$

$$\frac{(v,C,v') \in F \quad k \in \text{ThreadIDs} \quad C,(s,h,t) \rightsquigarrow_k q}{\mathsf{pc}[k:v],(s,h,t) \rightarrow_S \mathsf{pc}[k:v'],q} \quad (C \text{ is not } \textbf{fork} \text{ or } \textbf{join})$$

$$\frac{(v, x = \textbf{fork}(f), v') \in F \quad k,j \in \text{ThreadIDs}}{\mathsf{pc}[k:v],(s[x:(u,1)],h,t) \rightarrow_S \mathsf{pc}[k:v'][j:\mathsf{start}_f],(s[x:(j,1)],h,t[j:f])}$$

$$\frac{(v, x = \textbf{fork}(f), v') \in F \quad k \in \text{ThreadIDs} \quad \neg\exists u, s'.s = s'[x:(u,1)]}{\mathsf{pc}[k:v],(s,h,t) \rightarrow_S \mathsf{pc}[k:v'],\top}$$

$$\frac{(v, \textbf{join}(E), v') \in F \quad k \in \text{ThreadIDs} \quad \llbracket E \rrbracket_s{\downarrow}}{\mathsf{pc}[k:v][\llbracket E \rrbracket_s : \mathsf{end}_f],(s,h,t[\llbracket E \rrbracket_s : f]) \rightarrow_S \mathsf{pc}[k:v'],(s,h,t)}$$

$$\frac{(v, \textbf{join}(E), v') \in F \quad k \in \text{ThreadIDs} \quad \llbracket E \rrbracket_s{\uparrow} \vee (\llbracket E \rrbracket_s{\downarrow} \wedge t(\llbracket E \rrbracket_s){\uparrow})}{\mathsf{pc}[k:v],(s,h,t) \rightarrow_S \mathsf{pc}[k:v'],\top}$$

Fig. 10. Interleaving operational semantics in the case of dynamic thread creation. Here we reuse the relation $\rightsquigarrow_k$ defined in Figure 8 for $k \in \{\mathsf{M}\} \cup \text{ThreadIDs}$. Note that a thread is not deallocated until it is joined.

Note that the denotation of a formula P that has an empty lockset and contains no logical variables does not depend on the thread identifier and interpretation of logical variables with respect to which the formula is interpreted. We write denotations for such formulae simply as $\llbracket P \rrbracket$.

The soundness of the extended logic is stated in the following theorem.

Theorem 16 *Let S be the program* **let** $f_1 = C_1, \ \ldots, \ f_n = C_n$ **in** C *and* $\Gamma = \{P_{f_1}\} \ f_1 \ \{Q_{f_1}\}, \ \ldots, \ \{P_{f_n}\} \ f_n \ \{Q_{f_n}\}$*, where for all* $k = 1..n$ P_{f_k} *are precise and* P_{f_k}*,* Q_{f_k} *have an empty lockset. Suppose that*

$$\Gamma \vdash \{P_{f_1}\} \ C_1 \ \{Q_{f_1}\}, \ldots, \Gamma \vdash \{P_{f_n}\} \ C_n \ \{Q_{f_n}\}, \Gamma \vdash \{P\} \ C \ \{Q\}.$$

Then for any interpretation i *and state* $q_0 = (s_0, h_0, [\,])$ *such that* $q_0 \in \llbracket P \rrbracket_i^{\mathsf{M}} * \left(\circledast_{(A,u) \in \mathsf{Free}(q_0)} \mathcal{I}_0(A,u)\right)$ *the program* S *is safe when run from* q_0 *and if* $\mathsf{pc}_0, q_0 \rightarrow_S^* \mathsf{pc}, (s,h,t)$*, where* $\forall k.\mathsf{pc}(k){\downarrow} \Rightarrow \exists f.\mathsf{pc}(k) = \mathsf{end}_f$*, then* $(s,h,t) \in \llbracket Q \rrbracket_i^{\mathsf{M}} * \left(\circledast_{\{k \,|\, t(k){\downarrow}\}} \llbracket Q_{t(k)} \rrbracket\right) * \left(\circledast_{(A,u) \in \mathsf{Free}(s,h,t)} \mathcal{I}_0(A,u)\right)$.

It is interesting to note that the requirement that thread preconditions be precise in the theorem is similar to the requirement imposed to ensure the soundness of the hypothetical frame rule [12].

C.2 Proof of soundness

Here we adapt the proof of soundness given in Appendix B to the setting with dynamic thread creation.

Let $\mathcal{D}$ be the domain of states as defined in Section 8. We define a thread-local forward predicate transformer $\mathsf{Post}_k^{\gamma}(C) : \mathcal{D} \rightarrow \mathcal{D}$ for every thread k, command C, and *semantic procedure context* γ consisting of triples of the form $\{p\} \ f \ \{q\}$, where $p, q \subseteq$ States, p is precise, and p and q have an empty lockset. For commands C other than **fork** or **join** we let $\mathsf{Post}_k^\gamma(C) = \mathsf{Post}_k(C)$ as defined in Section B.1. Let

$$\mathsf{Post}_k^\gamma(x = \mathbf{fork}(f), (s, h, t)) = \{(s'[x : (j, 1)], h', t'[j : f]) \mid j \in \text{ThreadIDs}\},$$

if $\{p\}\ f\ \{q\} \in \gamma$, $\mathsf{rest}((s, h, t), p) = (s'[x : (u, 1)], h', t')$, and $\mathsf{Post}_k^\gamma(x = \mathbf{fork}(f), (s, h, t)) = \top$ otherwise. Let

$$\mathsf{Post}_k^\gamma(\mathbf{join}(E), (s, h, t)) = \{(s, h, t')\} * q,$$

if $\{p\}\ f\ \{q\} \in \gamma$, $[\![E]\!]_s\!\downarrow$, and $t = t'[[\![E]\!]_s : f]$, and $\mathsf{Post}_k^\gamma(\mathbf{join}(E), (s, h, t)) = \top$ otherwise. Here the requirement that preconditions in γ be precise ensures the determinism of splitting the state at **fork** commands. Post_k^γ for composite commands is defined as before with the aid of a functional $F_k(\gamma, C, p)$. The definition of validity with respect to the thread-local semantics now has to take into account procedure contexts. For a procedure context Γ we denote with $[\![\Gamma]\!]$ its corresponding semantic procedure context consisting of specifications $\{[\![P]\!]\}\ f\ \{[\![Q]\!]\}$, where $\{P\}\ f\ \{Q\}$ is in Γ.

Definition 17 *For a command C*

$$\Gamma \vDash_k \{P\}\ C\ \{Q\} \Leftrightarrow \mathsf{Post}_k^{[\![\Gamma]\!]}(C, [\![P]\!]^k) \sqsubseteq [\![Q]\!]^k.$$

Lemma 18 *If $\Gamma \vdash \{P\}\ C\ \{Q\}$, then for all $k \in$ ThreadIDs $\ \Gamma \vDash_k \{P\}\ C\ \{Q\}$.*

Proof. Lemmas 6 and 7 are still valid in the new setting. Besides, it is easy to show that the definition of Post_k^Γ implies the soundness of the rules Fork and Join with respect to the thread-local semantics (for this the requirement that thread pre- and postconditions have an empty lockset is essential). An analog of Lemma 8 can be stated and proved in the same way as before. We can then perform induction on the derivation of $\Gamma \vdash \{P\}\ C\ \{Q\}$ using the new lemmas. □

The Over-approximation Lemma is now formulated as follows.

Lemma 19 *Let S be the program* **let** $f_1 = C_1, \ \ldots, \ f_n = C_n$ **in** C *equipped with a precondition $p \in \mathcal{D}$ and a semantical procedure context $\gamma = \{p_1\}\ f_1\ \{q_1\}, \ldots, \{p_n\}\ f_n\ \{q_n\}$ such that $\mathsf{Post}_j^\gamma(C_k, p_k) \sqsubseteq q_k$ for all $k = 1..n$ and $j \in$ ThreadIDs. Let $g_{\mathsf{M}}(\mathsf{main}) = \mathsf{lfp}(F_{\mathsf{M}}(\gamma, C, p))$ and $g_j(f_k) = \mathsf{lfp}(F_j(\gamma, C_k, p_k))$ for all $k = 1..n$ and $j \in$ ThreadIDs. Then for any state $q_0 = (s_0, h_0, [\,])$ such that*

$$\{q_0\} \sqsubseteq p * \left(\mathop{\circledast}_{(A,u) \in \mathsf{Free}(q_0)} \mathcal{I}_0(A, u) \right) \tag{9}$$

whenever $\mathsf{pc}_0, q_0 \to_S^* \mathsf{pc}, q$, it is the case that

$$\{q\} \sqsubseteq \left(\mathop{\circledast}_{\{k \mid \mathsf{pc}(k)\downarrow\}} g_k(\mathsf{proc}(\mathsf{pc}(k)), \mathsf{pc}(k)) \right) * \left(\mathop{\circledast}_{(A,u) \in \mathsf{Free}(q)} \mathcal{I}_0(A, u) \right). \tag{10}$$

Proof. We prove the statement of the theorem by induction on the length m of the derivation of q. For $m = 0$ (10) is equivalent to (9). Suppose now that $\mathsf{pc}_0, q_0 \rightarrow_S^* \mathsf{pc}[j : v], q \rightarrow_S \mathsf{pc}'[j : v']$, $(v, C, v') \in F$, and

$$\{q\} \sqsubseteq \left(\underset{\{k \,|\, \mathsf{pc}(k)\downarrow\}}{\circledast} g_k(\mathsf{proc}(\mathsf{pc}(k)), \mathsf{pc}(k)) \right) * g_j(\mathsf{proc}(v), v) * \left(\underset{(A,u) \in \mathsf{Free}(q)}{\circledast} \mathcal{I}_0(A, u) \right).$$

We have to prove that

$$\{q'\} \sqsubseteq \left(\underset{\{k \,|\, \mathsf{pc}'(k)\downarrow\}}{\circledast} g_k(\mathsf{proc}(\mathsf{pc}'(k)), \mathsf{pc}'(k)) \right) * g_j(\mathsf{proc}(v'), v') * \left(\underset{(A,u) \in \mathsf{Free}(q')}{\circledast} \mathcal{I}_0(A, u) \right). \tag{11}$$

We consider three cases corresponding to the type of the command C.

1. C is not **fork** *or* **join**. In this case the proof is the same as in Theorem 11 modulo different notation. The notion of transfer functions for commands other than **fork** and **join** can be formulated in the same way as before and Lemma 12 still holds.

2. C is $x =$ **fork**(f_l). We can assume that $q = r_1 * r$, where $r_1 \in g_j(\mathsf{proc}(v), v)$, $r \in \left(\circledast_{\{k \,|\, \mathsf{pc}(k)\downarrow\}} g_k(\mathsf{proc}(\mathsf{pc}(k)), \mathsf{pc}(k))\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(q)} \mathcal{I}_0(A, u)\right)$, $r_1 = (s_1[x : (u, 1)], h_1, t_1)$, $\mathsf{pc}' = \mathsf{pc}[j' : \mathsf{start}_{f_l}]$, $q' = (s_1[x; (j', 1)], h_1, t_1[j' : f_l]) * r$, and $\mathsf{Post}_j^\gamma(C, r_1) \sqsubseteq g_j(\mathsf{proc}(v'), v')$; otherwise the right-hand side of (11) is $\top$. We then have to prove that $\{q'\} \sqsubseteq g_j(\mathsf{proc}(v'), v') * \{r\} * p_l$. We can assume that $\mathsf{rest}((s_1[x : (j', 1)], h_1, t_1), p_l) = (s'[x : (u, 1)], h', t')$; otherwise as $\mathsf{Post}_j^\gamma(C, r_1) \sqsubseteq g_j(\mathsf{proc}(v'), v')$, the right-hand side of (11) is $\top$. Let $r_2 = (s'[x : (j', 1)], h', t'[j' : f])$. Then $\mathsf{Post}_j^\gamma(C, r_1) \sqsubseteq g_j(\mathsf{proc}(v'), v')$ implies $\{r_2\} \sqsubseteq g_j(\mathsf{proc}(v'), v')$ and it is therefore sufficient to prove that $\{q'\} \sqsubseteq \{r_2\} * \{r\} * p_l$. It is easy to see that $\mathsf{rest}((s_1, h_1, t_1), p_l) = (s', h', t')$, hence, $\{(s_1, h_1, t_1)\} \sqsubseteq \{(s', h', t')\} * p_l$. Then $\{q'\} = \{(s_1[x : (j', 1)], h_1, t_1[j' : f_l])\} * \{r\} \sqsubseteq \{(s'[x : (j', 1)], h', t'[j' : f])\} * p_l * \{r\} = \{r_2\} * \{r\} * p_l$.

3. C is **join**(E). We can assume that $q = r_1 * r * r_2$, where $r_1 = (s_1, h_1, t_1[e : f_l])$, $\{r_2\} \sqsubseteq g_e(f_l, \mathsf{end}_{f_l})$, $r \in \left(\circledast_{\{k \,|\, \mathsf{pc}(k)\downarrow\}} g_k(\mathsf{proc}(\mathsf{pc}(k)), \mathsf{pc}(k))\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(q)} \mathcal{I}_0(A, u)\right)$, $\mathsf{pc} = \mathsf{pc}'[e : \mathsf{end}_{f_l}]$, $e = \llbracket E \rrbracket_{s_1}$, $q' = (s_1, h_1, t_1) * r * r_2$; otherwise either the left-hand side of (11) is $\emptyset$, or its right-hand side is $\top$. We have to prove that $\{q'\} \sqsubseteq g_j(\mathsf{proc}(v'), v') * \{r\}$. Since $\mathsf{Post}_e^\gamma(C_l, p_l) \sqsubseteq q_l$, by Definition 17 we have that $g_e(f_l, \mathsf{end}_{f_l}) \sqsubseteq q_l$, hence, $\{r_2\} \sqsubseteq q_l$. By the definition of Post_j^γ for **join**(E) we then get $\{(s_1, h_1, t_1)\} * q_l \sqsubseteq g_j(\mathsf{proc}(v'), v')$, hence, $\{q'\} = \{(s_1, h_1, t_1) * r_2 * r\} \sqsubseteq \{(s_1, h_1, t_1)\} * g_l * \{r\} \sqsubseteq g_j(\mathsf{proc}(v'), v') * \{r\}$.

So, in all cases (11) holds, which implies the statement of the theorem. □

We are now in a position to prove the soundness theorem for the extended logic.

Proof of Theorem 16. Consider an interpretation i and a state $q_0 = (s_0, h_0, [\,])$ such that $q_0 \in \llbracket P \rrbracket_i^{\mathsf{M}} * \left(\circledast_{(A,u)\in\mathsf{Free}(q_0)} \mathcal{I}_0(A, u)\right)$. Let $p = \llbracket P \rrbracket_i^{\mathsf{M}}$, $p_k = \llbracket P \rrbracket$, $q_k = \llbracket Q \rrbracket$, and $\gamma = \llbracket \Gamma \rrbracket$ in Lemma 19. By Lemma 18 for all $j \in$ ThreadIDs $\Gamma \vDash_j \{p_k\}\ C_k\ \{q_k\}$ and $\Gamma \vDash_{\mathsf{M}} \{p\}\ C\ \{q\}$, hence, by Definition 17, $\mathsf{Post}_j^\gamma(C_k, p_k) \sqsubseteq q_k$ and $\mathsf{Post}_{\mathsf{M}}^\gamma(C, p) \sqsubseteq q$. We have that (9) is fulfilled, which allows us to establish the safety of S. Suppose now that $\forall k.\mathsf{pc}(k)\downarrow \Rightarrow \exists f.\mathsf{pc}(k) = \mathsf{end}_f$ and $\mathsf{pc}_0, q_0 \rightarrow_S^* \mathsf{pc}, (s,h,t)$. From (10) we then get $\{(s,h,t)\} \sqsubseteq [\![Q]\!]_i^{\mathsf{M}} * \left(\circledast_{\{k \mid t(k)\downarrow\}} [\![Q_{t(k)}]\!]\right) * \left(\circledast_{(A,u)\in\mathsf{Free}(s,h,t)} \mathcal{I}_0(A,u)\right)$. □

C.3 Closure and admissibility

The pathological example of Section 7 can be modeled using thread handles and thread postconditions instead of lock handles and resource invariants: a postcondition of a thread can hold a handle for another thread and vice versa the main thread left without a handle for either of them. In this section we formulate conditions sufficient to rule out pathological situations involving both locks and threads and formulate a soundness statement that computes the set of final states of a program solely from postconditions of threads, without looking up the set of existing threads as done in Theorem 16.

For a state q let $\mathsf{Threads}(q)$ be the set of pairs from ThreadIDs × Procs consisting of thread identifiers and procedure names of threads existing in the state. We assume fixed a set of postconditions Q_f indexed by procedure names.

Definition 20 (Closure for programs with dynamic thread creation) *For $p \subseteq \text{States}$ let $c(p) \subseteq \text{States} \times \mathcal{P}(\text{ThreadIDs} \times \text{Procs})$ be the least predicate such that*

$$\{(q * r, \mathsf{Threads}(r)\backslash\mathsf{Threads}(q)) \mid (q,T) \in c(p) \wedge r \in \circledast_{(k,f)\in T} [\![Q_f]\!]\} \cup$$
$$\{(q * r, T \cup (\mathsf{Threads}(r)\backslash\mathsf{Threads}(q))) \mid (q,T) \in c(p) \wedge r \in \circledast_{(A,u)\in\mathsf{Unknown}(q)} \mathcal{I}_0(A,u)\} \cup$$
$$\{(q, \mathsf{Threads}(q)) \mid q \in p\} \subseteq c(p).$$

The closure $\langle p\rangle$ of p is the set of states from $\{q \mid (q,\emptyset) \in c(p)\}$ that do not contain locks with the value U.

Definition 21 (Admissibility of resource invariants and postconditions) *Resource invariants for a set of lock sorts $\mathcal{L}$ and postconditions Q_f for a set of procedures $\mathcal{F}$ are admissible if there do not exist sets $L \subseteq \mathcal{L} \times \text{Locs}$ and $T \subseteq \text{ThreadIDs} \times \mathcal{F}$ (such that $L \cup T \neq \emptyset$) and a state*

$$q \in \left(\circledast_{(A,u)\in L} \mathcal{I}_0(A,u)\right) * \left(\circledast_{(k,f)\in T} [\![Q_f]\!]\right)$$

such that for all $(A,u) \in L$ the permission associated with the lock at the address u in q is 1 and for all $(k,f) \in T$ the thread with the identifier k exists in q and is associated with the procedure f.

Theorem 22 *Let S be the program* **let** $f_1 = C_1, \ \ldots, \ f_n = C_n$ **in** C *and $\Gamma = \{P_{f_1}\}\ f_1\ \{Q_{f_1}\}, \ \ldots, \ \{P_{f_n}\}\ f_n\ \{Q_{f_n}\}$, where for all $k = 1..n$ P_{f_k} are precise and P_{f_k}, Q_{f_k} have an empty lockset. Suppose that*

$$\Gamma \vdash \{P_{f_1}\}\ C_1\ \{Q_{f_1}\}, \ldots, \Gamma \vdash \{P_{f_n}\}\ C_n\ \{Q_{f_n}\}, \Gamma \vdash \{P\}\ C\ \{Q\}.$$

Suppose further that either Q is intuitionistic or resource invariants for lock sorts used in the proofs and postconditions Q_{f_k} are admissible. Then for any interpretation i and complete state $q_0 = (s_0, h_0, [\,])$ such that $q_0 \in \langle [\![P]\!]_i^{\mathsf{M}} \rangle$ the program S is safe when run from q_0 and if $\mathsf{pc}_0, q_0 \rightarrow_S^* \mathsf{pc}, q$, where $\forall k. \mathsf{pc}(k)\downarrow \Rightarrow \exists f. \mathsf{pc}(k) = \mathsf{end}_f$, then $q \in \langle [\![Q]\!]_i^{\mathsf{M}} \rangle$.

Proof. Analogous to that of Theorem 4. □

Note that if the denotations of postconditions of all the threads of a program and resource invariants used in its proof have an empty thread pool, then we can use Definitions 2 and 3 instead of Definitions 20 and 21 in Theorem 22.

C.4 Parameter passing

We now consider the case when procedures representing the code of threads take parameters that are instantiated when threads are created:

$$\begin{array}{ll} C ::= \ldots \mid V = \mathbf{fork}(T, E) \mid \mathbf{join}(E) & \text{primitive commands} \\ P ::= \mathbf{let}\ T(V) = S,\ \ldots,\ T(V) = S\ \mathbf{in}\ S & \text{programs} \end{array}$$

To simplify notation we assume that each procedure has exactly one parameter. $\mathbf{fork}(f, E)$ now creates a new thread executing the code of the procedure f with the parameter E. Thread handles then have to carry information about the value E of the parameter with which the thread was instantiated: $\mathsf{tid}_f(F, E)$.

The design of proof rules that take into account parameter passing essentially follows proof rules for procedures with parameters in the case when variables are treated as resources [14]:

$$\frac{}{\Gamma \vdash \{P(X) * (O, x \Vdash \mathsf{emp_h} \land \mathsf{emp_t}) \land X = E\}\ x = \mathbf{fork}(f, E)\ \{O, x \Vdash \mathsf{emp_h} \land \mathsf{tid}_f(x, X)\}}\ \textsc{Fork}$$

$$\frac{}{\Gamma \vdash \{O \Vdash \mathsf{emp_h} \land \mathsf{tid}_f(E, X)\}\ \mathbf{join}(E)\ \{Q(X) * (O \Vdash \mathsf{emp_h} \land \mathsf{emp_t})\}}\ \textsc{Join}$$

where $\Gamma = \Gamma',\ \{P(X)\}\ f(X)\ \{Q(X)\}$ and Γ ranges over procedure contexts that now consist of specifications of the form $\{P(X)\}\ f(X)\ \{Q(X)\}$, where X is the only free logical variable in $P(X)$ and $Q(X)$.

A proof of the program $\mathbf{let}\ f_1(x) = C_1,\ \ldots,\ f_n(x) = C_n\ \mathbf{in}\ C$ is given by triples

$$\Gamma \vdash \{P_k(X) * (w \Vdash \mathsf{emp_h} \land \mathsf{emp_t}) \land w = X\}\ C_k[w/x]\ \{Q_k(X) * (w \Vdash \mathsf{emp_h} \land \mathsf{emp_t})\}$$

(where w is globally fresh) for $k = 1..n$ and $\Gamma \vdash \{P\}\ C\ \{Q\}$, where

$$\Gamma = \{P_1(X)\}\ f_1(X)\ \{Q_1(X)\},\ \ldots,\ \{P_n(X)\}\ f_n(X)\ \{Q_n(X)\}.$$

The semantical development carried out in this section can be adjusted to accommodate parameter passing. The same conditions required for soundness as before apply.

We can now give an example of using the proof rules for dynamic thread creation. The main thread of the program in Figure 11 allocates an array of n objects of the type *DATA* and creates n threads to process the objects. It then waits for the termination of all the threads and deallocates the array.

```
$\{\mathsf{emp}_{\mathbf{s}} \wedge \mathsf{emp}_{\mathbf{t}} \wedge D \mapsto \_\}$ $process(D)$ $\{\mathsf{emp}_{\mathbf{s}} \wedge \mathsf{emp}_{\mathbf{t}} \wedge D \mapsto \_\}$
process(DATA *d) {
  $\{d \Vdash \mathsf{emp}_{\mathbf{t}} \wedge d \mapsto \_\}$
  // ...Process d...
  $\{d \Vdash \mathsf{emp}_{\mathbf{t}} \wedge d \mapsto \_\}$
}

main(unsigned n) {
  DATA *x;
  THREAD_ID *t;
  int i;
  THREAD_ID id;

  $\{O \Vdash \mathsf{emp}_{\mathbf{h}} \wedge \mathsf{emp}_{\mathbf{t}}\}$
  x = new DATA[n];
  t = new THREAD_ID[n];
  $\{O \Vdash (\circledast_{K=0}^{n-1} (t + K \mapsto \_)) * (\circledast_{K=0}^{n-1} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}}\}$
  for (i = 0; i < n; i++) {
    $\{O \Vdash (\circledast_{K=0}^{i-1} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K))) *$
     $((\circledast_{K=i}^{n-1} (t + K \mapsto \_)) * (\circledast_{K=i}^{n-1} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}}) \wedge i < n\}$
    id = fork(process, x+i);
    $\{O \Vdash (\circledast_{K=0}^{i-1} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K))) *$
     $((\circledast_{K=i}^{n-1} (t + K \mapsto \_)) * (\circledast_{K=i+1}^{n-1} (x + K \mapsto \_)) \wedge \mathsf{tid}_{process}(id, x + i))\}$
    t[i] = id;
    $\{O \Vdash (\circledast_{K=0}^{i} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K))) *$
     $((\circledast_{K=i+1}^{n-1} (t + K \mapsto \_)) * (\circledast_{K=i+1}^{n-1} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}})\}$
  }
  $\{O \Vdash (\circledast_{K=0}^{n-1} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K)))\}$
  for (i = 0; i < n; i++) {
    $\{O \Vdash (\circledast_{K=i}^{n-1} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K))) *$
     $((\circledast_{K=0}^{i-1} (t + K \mapsto \_)) * (\circledast_{K=0}^{i-1} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}}) \wedge i < n\}$
    id = t[i];
    $\{O \Vdash (\circledast_{K=i+1}^{n-1} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K))) *$
     $(t + i \mapsto id \wedge \mathsf{tid}_{process}(id, x + i)) * ((\circledast_{K=0}^{i-1} (t + K \mapsto \_)) * (\circledast_{K=0}^{i-1} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}})\}$
    join(id);
    $\{O \Vdash (\circledast_{K=i+1}^{n-1} (\exists T. t + K \mapsto T \wedge \mathsf{tid}_{process}(T, x + K))) *$
     $((\circledast_{K=0}^{i} (t + K \mapsto \_)) * (\circledast_{K=0}^{i} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}})\}$
  }
  $\{O \Vdash (\circledast_{K=0}^{n-1} (t + K \mapsto \_)) * (\circledast_{K=0}^{n-1} (x + K \mapsto \_)) \wedge \mathsf{emp}_{\mathbf{t}}\}$
  delete[n] x;
  delete[n] t;
  $\{O \Vdash \mathsf{emp}_{\mathbf{h}} \wedge \mathsf{emp}_{\mathbf{t}}\}$
}
```

Fig. 11. Proof outline for a program with dynamic thread creation. Here O is n, x, t, i, id.