Proof-Directed Parallelization Synthesis by Separation Logic

MATKO BOTINČAN and MIKE DODDS, University of Cambridge
SURESH JAGANNATHAN, Purdue University

We present an analysis which takes as its input a sequential program, augmented with annotations indicating potential parallelization opportunities, and a sequential proof, written in separation logic, and produces a correctly synchronized parallelized program and proof of that program. Unlike previous work, ours is not a simple independence analysis that admits parallelization only when threads do not interfere; rather, we insert synchronization to preserve dependencies in the sequential program that might be violated by a naïve translation. Separation logic allows us to parallelize fine-grained patterns of resource usage, moving beyond straightforward points-to analysis. The sequential proof need only represent shape properties, meaning we can handle complex algorithms without verifying every aspect of their behavior.

Our analysis works by using the sequential proof to discover dependencies between different parts of the program. It leverages these discovered dependencies to guide the insertion of synchronization primitives into the parallelized program, and to ensure that the resulting parallelized program satisfies the same specification as the original sequential program, and exhibits the same sequential behavior. Our analysis is built using frame inference and abduction, two techniques supported by an increasing number of separation logic tools.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*Correctness proofs*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Separation logic, abduction, frame inference, deterministic parallelism

ACM Reference Format:
Botinčan, M., Dodds, M., and Jagannathan, S. 2013. Proof-directed parallelization synthesis by separation logic. *ACM Trans. Program. Lang. Syst.* 35, 2, Article 8 (July 2013), 60 pages.
DOI: http://dx.doi.org/10.1145/2491522.2491525

8

1. INTRODUCTION

In many cases concurrency is an optimization, rather than intrinsic to the functional behavior of a program. That is, a concurrent program is often intended to achieve the same effect of a simpler sequential counterpart, but faster (e.g., improved response time). Error-free concurrent programming remains a tricky problem, beyond the capabilities of most programmers; consequently, an attractive alternative to manually synchronizing a concurrent program is to automatically *synthesize* one. In this approach,

This work was supported by the Gates trust, by EPSRC grants EP/H010815/1, EP/H005633/1, EP/F036345 and by NSF grant CCF-0811631.
M. Dodds is currently affiliated with the University of York.
Authors' addresses: M. Botinčan, Computer Laboratory, University of Cambridge; email: matko.botincan@gmail.com; M. Dodds, Department of Computer Science, University of York, UK; S. Jagannathan, Department of Computer Science, Purdue University.

DOI: http://dx.doi.org/10.1145/2491522.2491525

the programmer writes a sequential program, which is then automatically transformed into a concurrent one exhibiting the same behavior. Programmers can work in the simpler, more reliable sequential setting, yet reap the performance benefit of concurrency.

Making a program concurrent may also make it more complex to verify its correctness. To be useful in practice, a concurrent verification tool must (a) explore a potentially large number of interleavings, and (b) construct precise flow- and path-sensitive abstractions of a shared heap. This complexity is often at odds with the straightforward intentions of the programmer, expressed in the original sequential programs. Consequently, the verification problem is rendered significantly more tractable by synthesizing concurrency. Understanding and verifying a concurrent program then reduces to first verifying the sequential program, and second, verifying the parallelizing transformation.

We propose a program analysis and transformation that automatically yields a parallelized program given a sequential one. Our approach is built on verification technology; our analysis is guided by a lightweight proof of the sequential program, written in separation logic. We assume that the programmer annotates points in the sequential program where concurrency might be profitably exploited, without supplying any additional concurrency control or synchronization. The result of the transformation is a concurrent program with corresponding behavior, and a safety proof for the concurrent program; in this sense, parallelized programs are verified by construction.

Our transformation ensures that the input-output behavior of the sequential program is preserved by requiring that the concurrent program respects sequential data dependencies. In other words, the way threads access and modify shared resources never results in behavior that would not be possible under sequential evaluation. To enforce these dependencies, the transformation injects *synchronization barriers*, signaling operations that regulate when threads are allowed to read or write shared state. These barriers can be viewed as resource transfer operations that acquire and relinquish access to shared resources such as shared-memory data structures and regions when necessary.

Our analysis is built on separation logic [Reynolds 2002]. The input safety proof is used to discover how resources are demanded and consumed within the program, and to synthesize barriers to precisely control access to these resources. Our approach relies on frame inference [Berdine et al. 2005b] and abduction [Calcagno et al. 2011], two techniques that generate fine-grained information capable of describing when resources are necessary and when they are redundant. This information enables optimizations that depend on deep structural properties of the resource; for example, we can split a linked list into dynamically sized segments and transmit portions between threads piece-by-piece.

The input proof supplied to our analysis must state all resources that are used by the program such as the *shape* of all data-structures (for example, that a data-structure is a linked list). However, it need not establish the functional correctness of the algorithm. Our analysis protects all dependencies that are relevant to the program's input-output behavior, not just those specified in the proof. Thus we can apply our approach to complex algorithms without verifying every aspect of the program's behavior.

Our analysis thus enforces sequential order over visible input-output behaviors, while allowing parallelization when it would have no visible effect. Insofar as our technique safely transforms a sequential program into a concurrent one, it can also be viewed as a kind of proof-directed compiler optimization.

Contributions.

(1) We present an automated technique to synthesize a parallel program given a partially specified sequential program augmented with annotations indicating

computations that are candidates for parallelization. The provided specifications are used to define relevant loop invariants.

(2) We leverage abduction and frame inference to define a path- and context-sensitive program analysis capable of identifying per program-point resources that are both *redundant*—resources that would no longer be used by the thread executing this computation; and, *needed*—resources that would be required by any thread that executes this computation, but which are not known to have been released at this point.

(3) We use information from the above analysis to inject *grant* and *wait* barriers into the original program; their semantics enable resource transfer from redundant to needed resource points. The analysis also constructs a safety proof in separation logic which validates the correctness of the barrier injections.

(4) We prove that our transformation is specification-preserving: the parallelized program is guaranteed to satisfy the same specification as the sequential program. Moreover, for terminating programs that do not dispose memory, we also show that the transformed program preserves the *behavior* of the original.

Article structure. An overview of the approach and motivating examples are given in Section 2. Extension for dealing with loops and recursive data structures (such as lists) are discussed in Section 3. Technical details of the analysis are given Section 4, Section 5, and Section 6. Observations about the way the choice of parameters affects the analysis are given in Section 7. Limitations of the analysis are discussed in Section 8. A formal semantics is given in Section 9 and the soundness results—behavior preservation and termination—are presented in Section 10 and Section 11. Section 12 discusses related work.

2. OVERVIEW

Our analysis takes a sequential program, annotated to indicate segments that might be feasible candidates to execute in parallel, and produces a parallelized program. This new program should be observationally equivalent to the sequential original; that is, it should have identical input-output behavior. To ensure this, we execute all the identified segments in parallel, but insert sufficient synchronization barriers to enforce those dependencies that can affect the input-output behavior of the program.

In a sequential program execution, a *dependency* exists between two events if the effect of one event can affect the behavior of a subsequently executed one; these effects can be based on control-flow (e.g., the outcome of a conditional expression) or data-flow (e.g., assigning a value to a location that is subsequently read). If new dependencies are introduced or removed during parallelization, the input-output behavior of the program may change. Our analysis uses *separation logic* to discover sequential dependencies, and to ensure that they are preserved by parallelization.

We assume that the original program is accompanied by a proof, written in sequential separation logic. The proof need not establish full functional correctness; it suffices that it establishes that the program does not fault (e.g., does not access a data structure incorrectly). This proof is used to drive the parallelization process, allowing our analysis to calculate precisely which statements access what resources (i.e., mutable state such as data structures or components thereof). Intuitively, once our analysis detects that a resource will not be accessed for the remainder of a program segment, it can be safely accessed concurrently without adding or removing dependencies. Until this point, no other program statements that access this resource may run concurrently in any transformed parallelized version of the program.

2.1. Overview of the Parallelization Process

In the simplest scenario, the input sequential program is just a sequential composition $C_1; C_2$ of two subprograms C_1 and C_2. (Later we will extend our analysis to more complex target programs). We assume the programmer has identified C_1 and C_2 as candidates for parallelization. The intended semantics is that C_1 and C_2 may run in parallel, but that the visible behavior will be identical to running them in sequence; this semantics provides a simple but useful form of data parallelism in which concurrently executing computations may nonetheless access shared state.

To ensure behavior preservation, we often need to insert synchronization operations. For example, consider the following program:

```
f(1); f(2),
```

where `f()` is a procedure manipulating shared locations `x` and `y`.

```
f(int i) {
  int v = *x;
  if (v>=i) *y = v;
  else *x = 0;
}.
```

How can we parallelize this program without introducing any unwanted new behaviors, that is, behaviors not possible under sequential execution? Naïvely, we might simply run both calls in parallel, without synchronization. That is:

```
f(1) || f(2).
```

In some situations this parallelization is good, and introduces no new observable behaviors (e.g., if the initial value stored at `x` is `0`). But, under others, the second call to `f()` may write to a memory location that is subsequently read by the first call to `f()`, violating the intended sequential ordering. For example, consider the case when the value stored at `x` is initially `1`; sequential execution would produce a final result in which the locations pointed to by `x` and `y` resp. are `0` and `1`, while executing the second call to completion before the first would yield a result in which the location pointed to by `y` remains undefined.

We say one portion of a parallelized program occurs *logically earlier* than another if the two portions were sequentially ordered in the original source program. In the above example, the call `f(1)` occurs logically earlier than `f(2)`. To be sure that no new observable behavior is introduced during a parallelization transformation, we must ensure that:

— a computation cannot read from a location that was already written to by a computation that occurs logically later; and
— a computation cannot write to a location that was already read from or written to by a computation that occurs logically later.

Note, crucially, that a computation that occurs logically later from another need not always wait for the earlier one—synchronization is only needed when reads and writes to a particular memory location could result in out-of-order behavior. In other words, we want to enforce only salient dependencies, while allowing beneficial race-free concurrency.

In this section, we considered parallelizing only a single sequentially composed pair of program segments that access fixed memory locations. Later we will consider parallel-for loops that execute all iterations concurrently (Section 3.1), over dynamic data-structures (Section 3.2).

```
f1(i){
 int v = *x;
 if (v>=i) {
   grant(wx);
   *y = v;
   grant(wy);
 }
 else {
   grant(wy);
   *x = 0;
   grant(wx);
 }
}
```

```
f2(i){
 wait(wx);
 int v = *x;
 if (v>=i) {
   wait(wy);
   *y = v;
 }
 else {
   *x = 0;
   wait(wy);
 }
}
```

Fig. 1. Parallelization of the two calls to `f()`.

2.2. Dependency-Enforcing Barriers

In order to enforce the logical ordering in the parallelized program, our analysis inserts barriers enforcing the logical order over resources, giving the modified procedures $f_1()$ and $f_2()$. The safely parallelized program is then:

```
f1(1) || f2(2).
```

To enforce the ordering between calls to `f()`, we introduce `grant()`, a barrier that signals that the resource it protects can safely be accessed by a computation that occurs logically afterwards; and `wait()`, a barrier that blocks until the associated resource becomes available from a computation that occurs logically before it [Dodds et al. 2011].

How do we use these barriers to enforce sequential dependencies? The exact pattern of parallelization depends on the granularity of our parallelization analysis. In the best case, there are no dependencies between successive calls (e.g., each invocation operates on a different portion of a data structure). In this case, we need no barriers, and all invocations run independently. However, our example program shares the locations `x` and `y`, meaning barriers are required.

In the simplest parallelization of our example, the call to $f_1()$ would end with a call to `grant()`, while the call to $f_2()$ would begin with a call to `wait()`—this would enforce a total sequential ordering on the invocations. A better (although still relatively simple) parallelization is shown in Figure 1. Two channels, `wx` and `wy`, are used to mediate signaling between f_1 and f_2—`wx` (resp. `wy`) is used to signal that the later thread can read and write to the heap location referred to by `x` (resp. `y`).

Our analysis inserts a `grant()` barrier when the associated resource will no longer be accessed. Similarly, it injects a `wait()` barrier to wait for the resource to become available before it is accessed.

How does our analysis generate these synchronization barriers? The example we have given here is simple, but in general our analysis must cope with complex dynamically allocated resources such as linked lists. It should deal with partial ownership (for example, read-access), and with manipulating portions of a dynamic structure (e.g., just the head of a linked list, rather than the entire list). We therefore need a means to express complex, dynamic patterns of resource management and transfer.

To achieve this, our analysis assumes a sequential *proof*, written in separation logic, rather than just an undecorated program. This proof need not capture full functional correctness: it is sufficient just to prove that the program does not fault. We exploit the dependencies expressed within this proof to determine the resources that are needed at a program point, and when they can be released. Our analysis inserts barriers to enforce the sequential dependencies represented in this proof. As our proof system

is sound, these dependencies faithfully represent those in the original sequential program.

2.3. Resources, Separation and Dependency

Our analysis works by calculating the ways that resources are used in the source sequential program. By *resource*, we mean mutable state that can generates dependencies between program statements. For example: heap cells in memory; objects built from heap cells such as linked lists; portions of objects, such as linked-list segments; and other kinds of mutable objects such as channels. If two program statements access the same resource, there may be a dependency between them. If they do not, then a dependency cannot exist.

At the heart of our analysis is automated reasoning using separation logic [O'Hearn 2007; Reynolds 2002]. Separation logic is a Hoare-style partial correctness logic for reasoning about mutable resources. It allows us to represent the use of resources in the sequential program, and to soundly manipulate the flow of resources to give a parallelized program.

The power of separation logic lies in its ability to cleanly represent and compose resources. Indeed, resources in our analysis are simply separation logic assertions. Two assertions can be composed using the $*$ operator (the *separating conjunction*) to give a larger assertion. An assertion $P * F$ is satisfied if P and F hold, and the resources denoted do not overlap. In the case of heap resources, separation by $*$ corresponds to disjointness of heap address spaces.

As well as establishing the absence of faults, judgements in separation logic also establish the resource bounds of a program. A judgement $\{P\}\ C\ \{Q\}$ can be be read as saying: "if C is run in a resource satisfying P, it (1) will not fault, (2) will give a resource satisfying Q if it terminates, and (3) will only access resources held in P or acquired through resource transfer".

This property—that all resources be described in preconditions or acquired explicitly—is an essential feature of separation logic (it is sometimes called the *tight interpretation* of specifications). A judgement $\{P\}\ C\ \{Q\}$ means that C does not access any resources not described in P (conversely, P can include resources that C does not access). For example, the following judgement is invalid in separation logic.

$$\{\mathtt{x} \mapsto v_1'\} * y = 5\ \{\mathtt{x} \mapsto v_2'\}$$

(We write $a \mapsto b$ to indicate that the address a maps to the value b in the heap. By convention, primed variables are implicitly existentially quantified.) This judgement is invalid because `x` and `y` might point to different locations, meaning the program might write to a resource not mentioned in its precondition. In classical Hoare logic this judgement would be valid.

The tight interpretation is essential for the soundness of our analysis. Suppose we prove a specification for a program (or portion thereof); resources outside of the precondition cannot be accessed by the program, and cannot affect its behavior. If two program statements access disjoint resources, then there can be no dependencies between the two statements.[1] – consequently such resources can be safely transferred to other program segments that are intended to be executed in parallel.

[1]However, there is an important caveat regarding memory allocation and disposal – see Section 6 for a discussion.

The key proof rule of separation logic is the *frame rule*, which allows small specifications to be embedded into a larger context.

$$\frac{\{P\}\ C\ \{Q\}}{\{P * F\}\ C\ \{Q * F\}}\ \textsc{Frame}$$

The concurrent counterpart of the frame rule is the *parallel rule*. This allows two threads that access non-overlapping resources to run in parallel, without affecting each other's behavior.

$$\frac{\{P_1\}\ C_1\ \{Q_1\} \qquad \{P_2\}\ C_2\ \{Q_2\}}{\{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}\ \textsc{Parallel}$$

The parallel rule enforces the absence of data-races between C_1 and C_2. The two threads can consequently only affect each other's behavior by communicating through race-free synchronization mechanisms, such as locks or barriers, that act as shared resources.

Assertion language. As well as the separating conjunction $*$ and the standard first-order logical operators, $\vee$, $\wedge$, $\exists$, we assume a number of basic assertions for representing heap states. The simplest is emp, which asserts that the local heap is empty. We have already encountered the points-to assertion $a \mapsto v$, which says that the heap cell with address a holds value v. Assertions in separation logic represent ownership, so this assertion also says that the thread is allowed to read and write to address a freely. Where the object at an address can have more than one field, we write $a.f \mapsto v$ to represent an individual field f. By convention, primed variables are implicitly existentially quantified.

We also assume a *fractional* points-to assertion in the style of [Bornat et al. 2005] to allow a location to be shared between threads. An assertion $a \overset{f}{\mapsto} v$ for some $f \in (0, 1]$ means that the current thread holds permission to read f on address a. If $f = 1$, the current thread also holds permission to write to the address. Fractional permissions can be split and recombined according to the following rules.

$$a \overset{f_1+f_2}{\longmapsto} v \iff a \overset{f_1}{\mapsto} v * a \overset{f_2}{\mapsto} v \qquad\qquad a \mapsto v \iff a \overset{1}{\mapsto} v$$

Using fractional permissions, read-only access can be shared between several threads in the system. Read-write permission can be recovered if some thread collects together all of the fractional permissions for the address.

To represent lists, we assume a predicate $\mathsf{lseg}(x, t)$, which asserts that a segment of a linked list exists with head x and tail-pointer t. We write $\mathsf{node}(x, y)$ to represent a single node in the list:

$$\mathsf{node}(x, y) \quad \triangleq \quad x.\mathrm{val} \mapsto v' * x.\mathrm{nxt} \mapsto y.$$

We then define lseg as the least separation logic predicate satisfying the following recursive equation:

$$\mathsf{lseg}(x, t) \quad \triangleq \quad (x = t \wedge \mathsf{emp}) \vee (\mathsf{node}(x, y') * \mathsf{lseg}(y', t)).$$

The behavior of the analysis depends strongly on the choice of these resource predicates. We discuss alternatives to lseg, node, etc. in Section 7.

Automatic inference. Automated reasoning and symbolic execution in separation logic revolves around two kinds of inference questions: frame inference and abduction. These questions form the basis of symbolic execution in separation logic [Berdine et al. 2005a; Calcagno et al. 2011]. Intuitively, frame inference lets us reason forwards, while abduction lets us reason backwards.

The first question, *frame inference*, calculates the portion of an input assertion which is 'left over', once a desired assertion has been satisfied. Given an input assertion S and desired assertion P, an assertion $F_?$ is a valid frame if $S \vdash P * F_?$. We write the inference question as follows.

$$S \vdash P * [F_?]$$

(Throughout the article, we use square brackets to indicate the portions of an entailment that should be computed.)

Frame inference is used during forwards symbolic execution to calculate the effects of commands. Suppose we have a symbolic state represented by an assertion S, and a command c with specification $\{P\}\, c\, \{Q\}$. If we calculate the frame in $S \vdash P * [F_?]$, the symbolic state after executing c must be $Q * F_?$. For example, consider the following command and specification:

$$\{\mathtt{x} \mapsto v'\} * \mathtt{x} = 42\ \{\mathtt{x} \mapsto 42\}.$$

With initial symbolic state $S = \mathtt{x} \mapsto 5 * \mathtt{y} \mapsto 7$ and desired precondition $P = \mathtt{x} \mapsto v'$ we obtain the frame $\mathtt{y} \mapsto 7$. By combining this with the command's postcondition, we get the symbolic state after executing the command, $\mathtt{x} \mapsto 42 * \mathtt{y} \mapsto 7$.

The second kind of inference question, *abduction*, calculates an *antiframe*—the "missing" assertion that must be combined with an input assertion in order to satisfy some desired assertion. Given an input assertion S and a desired assertion P, an assertion $A_?$ is a valid antiframe if $S * A_? \vdash P$. We write the inference question as follows:

$$S * [A_?] \vdash P.$$

Abduction is used during a backwards analysis to calculate what extra resources are necessary to execute the command safely. Suppose we have a symbolic state represented by an assertion S, and a command c with specification $\{P\}\, c\, \{Q\}$. It might be that S doesn't include sufficient resources to execute c. In this case, if we calculate the antiframe $S * [A_?] \vdash P$, the assertion $S * A_?$ will be sufficient to execute c safely. For example, suppose we have the following command and specification:

$$\{\mathtt{x} \mapsto v'_1 * \mathtt{y} \mapsto v'_2\}\ \mathtt{if}\ (*\mathtt{y}{==}0)\ *x = 42\ \{\mathtt{x} \mapsto v'_3 * \mathtt{y} \mapsto v'_4\}.$$

With symbolic state $S = \mathtt{x} \mapsto 5 * \mathtt{z} \mapsto 7$ and desired precondition $P = \mathtt{x} \mapsto v'_1 * \mathtt{y} \mapsto v'_2$, abduction will obtain the antiframe $A_? = \mathtt{y} \mapsto v'$. The combined assertion $\mathtt{x} \mapsto 5 * \mathtt{z} \mapsto 7 * \mathtt{y} \mapsto v'$ suffices to execute the command safely.

The tight interpretation of specifications is necessary for this kind of reasoning. Because a specification must describe all the resources affected by a thread, any resources in the frame must be unaccessed. Conversely, if we calculate the antiframe $S * [A_?] \vdash P$, it must be the case that before executing c, we must first acquire the additional resource $A_?$ (as well as S).

Redundant and needed resources. In a separation logic proof, the assertion at a particular program point may describe resources that are not needed until much later in the program (or that are not needed by any subsequent command). Different commands access different resources, but the tight interpretation means that each assertion expresses at least the resources used by the program at any point during its execution (aside from those gained and lost by resource transfer). Thus at a given point in the program, many resources that are represented in the proof will be redundant.

Resources that become redundant will not be accessed by the current thread, and thus can be accessed by a parallel thread without generating data-races. Redundant resources can be seen as overapproximating independence: a resource that is

redundant at a particular point in a thread cannot subsequently generate dependencies between it and other threads. Our parallelization analysis uses the input separation logic proof to identify resources that are redundant, and uses `grant()` and `wait()` signaling to transfer them to the logically next point at which they are needed.

Thus our analysis is likely to be most effective for programs with complex specifications encompassing many resources, and with programs which modify resources in very few places, and where the resources accessed change over the course of the program. The `move()` example we discuss in Section 3.2 has precisely these characteristics.

We use frame inference to determine *redundant resources*—resources that will not be accessed in a particular portion of the program, and which can thus be safely transferred to other threads. Conversely, we use abduction to determine *needed resources*—resources that must be held for a particular portion of the program to complete, and which must thus be acquired before the program can proceed safely.

In our analysis, we calculate the redundant resource from the current program point to the end of the current parallelized program segment. This resource can be transferred to the logically next program segment with a `grant()` barrier. We calculate the needed resource from the start of current segment to the current program point. This resource must be acquired from the logically preceding segment using a `wait()` barrier before execution can proceed further.

Note that these two resources need not be disjoint. A resource may be used early in a segment, in which case it will be both needed up to the current point, and redundant from the current point. Note also that redundant and needed resources need not cover the entire state—some resource described in the proof may never be accessed or modified by the program.

2.4. Algorithm Overview

We assume the user supplies a sequential program with parallelizable segments identified, as well as a sequential proof written in separation logic establishing relevant memory safety properties of the program. The high-level structure of our algorithm is as follows.

(1) A *resource usage analysis* uses abduction and frame inference to discover redundant and needed resources for different portions of the program.
(2) A *parallelizing transformation* leverages this information to construct a parallelized program and associated separation logic proof of correctness:
   (a) The *parallelization* phase converts the sequential program into a concurrent program by replacing all the identified parallelizable segments with parallel threads.
   (b) The *resource matching* phase matches redundant resources in each parallel thread with needed resources in the logically following parallel thread.
   (c) The *barrier insertion* phase modifies each parallel thread to insert `grant()` and `wait()` barriers consistent with the discovered resource-transfer.

Algorithm properties. Our algorithm does not guarantee optimal parallelization; there may be programs that it does not find that are more parallel. However, the resulting parallelized program is guaranteed to be race-free and memory-safe, and the analysis constructs a separation logic proof that the parallelized program satisfies the same specification as the original sequential program. This new proof differs from the proof of the original sequential proof, reflecting parallelization and injected synchronization. If the original sequential program is terminating and does not dispose memory, then parallelization is behavior-preserving: every input-output

```
1  {x ↦ a * y ↦ b}
2  void f(i) {
3    int v = *x;
4   {v = a ∧ x ↦ a * y ↦ b}
5    if (v >= i) {
6     {v = a ∧ v ≥ i ∧ x ↦ a * y ↦ b}
7      *y = v;
8    }
9    else {
10    {v = a ∧ v < i ∧ x ↦ a * y ↦ b}
11     *x = 0;
12   }
13 }
14 {(a ≥ i ∧ x ↦ a * y ↦ a) ∨ (a < i ∧ x ↦ 0 * y ↦ b)}
```

Fig. 2. Separation logic proof of the function `f()`.

behavior of the parallelized program is also a behavior of the original sequential program.

2.5. Resource Usage Analysis

Consider once again the program that we introduced at the start of this section:

```
f(1); f(2)
```

and the following specification for `f()`, covering any input value `i`:

$$\{\mathtt{x} \mapsto a * \mathtt{y} \mapsto b\} \quad \mathtt{f(i)} \quad \{(a \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto a) \vee (a < \mathtt{i} \wedge \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto b)\}.$$

The precondition of this specification says that `f(i)` accesses two distinct memory locations, `x` and `y`. As is standard in Hoare logic, we use ghost variables a and b to record the values stored in `x` and `y`. The postcondition gives two possibilities: if a is greater than `i`, then both `x` is unmodified and `y` are set to a; otherwise, `x` is set to 0 and `y` unmodified. Figure 2 shows a proof of this specification.

Needed resources. At the core of the analysis is computing *needed* resources between pairs of program points, that is, the resource that must be held to execute from the start of the interval to the end without faulting. Needed resources are used to work out when `wait`-barriers must be injected. Resources must be acquired before they are used in a program segment. Thus, needed resources are calculated from the start of the segment, *to* every other program point.

In our example, `wait()` barriers are injected into the second call to `f()`. These needed resources are shown in Figure 3. For example, the resource calculated for the start of the `else`-branch (Figure 3, line 11) is

$$a < \mathtt{i} \wedge \mathtt{x} \mapsto a\,.$$

That is, to reach this program point from the start of `f()`, the thread only requires access to the heap cell `x`, provided $a < \mathtt{i}$. Access to `x` is needed because it is dereferenced in line 3.

Redundant resources. To calculate where `grant()` barriers can be inserted, the analysis computes *redundant* resources for particular program intervals. The redundant resource in an interval is the portion of the currently held resource that is *not* needed to execute to end of the interval. Resources that are redundant to the end of the segment can be transferred to logically later program segments using `grant()`.

```
1  n:{emp}
2  void f(i) {
3    int v = *x;
4    n:{x ↦ a}
5    if (v >= i) {
6      n:{a ≥ i ∧ x ↦ a}
7      *y = v;
8      n:{a ≥ i ∧ x ↦ a ∗ y ↦ b}
9    }
10   else {
11     n:{a < i ∧ x ↦ a}
12     *x = 0;
13     n:{a < i ∧ x ↦ a}
14   }
15 }
16 n:{(a ≥ i ∧ x ↦ a ∗ y ↦ b) ∨ (a < i ∧ x ↦ a)}
```

Fig. 3. Needed resources from the start of `f()` to all other program points.

Consequently, the redundant resource is calculated from each program point, to the end of the segment.

To calculate a redundant resource, the analysis first calculates the needed resource for the interval. It then subtracts the needed resource from the resource described in the sequential proof.

The redundant resources, along with the needed resources used to calculate them, for the first call to `f()` are shown in Figure 4. For example, the needed resource at the start of the `else`-branch (Figure 4, line 11) is

$$\mathtt{x} \mapsto a.$$

According to the sequential proof in Figure 2, such a thread actually holds the following resource at this point:

$$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b.$$

To calculate the redundant resource, we pose the following frame inference question, computing the frame $F_?$.

$$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \;\vdash\; \mathtt{x} \mapsto a * [F_?]\,.$$

The resulting redundant resource is as follows:

$$F_?\colon \quad \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto b.$$

In other words, the thread no longer requires access to the heap cell `y` beyond line 11 in `f()`, and we also know that `v` < `i`. Note however, that we do not know that `v` is the same as the value pointed to by `x`, because this fact can change later in the segment.

2.6. Parallelizing Transformation

As just described, the parallelizing transformation has three phases. First, the program is divided into two parallel threads.

```
f1(1) || f2(2)
```

The transformation specializes `f()` into distinct functions $\mathtt{f}_1\mathtt{()}$ and $\mathtt{f}_2\mathtt{()}$ because each thread has different synchronization.

```
1   n:{x ↦ a ∗ (a ≥ i ∧ y ↦ b)}
    r:{emp}
2   void f(i) {
3     int v = *x;
4     n:{(v ≥ i ∧ y ↦ b) ∨ (v < i ∧ x ↦ a)}
      r:{(v = a ∧ v ≥ i ∧ x ↦ a) ∨ (v < i ∧ y ↦ b)}
5     if (v >= i) {
6       n:{y ↦ b}
        r:{v = a ∧ v ≥ i ∧ x ↦ a}
7       *y = v;
8       n:{emp}
        r:{v = a ∧ v ≥ i ∧ x ↦ a ∗ y ↦ b}
9     }
10    else {
11      n:{x ↦ a}
        r:{v < i ∧ y ↦ b}
12      *x = 0;
13      n:{emp}
        r:{v = a ∧ v < i ∧ x ↦ a ∗ y ↦ b}
14    }
15  }
16  n:{emp}
    r:{x ↦ a ∗ y ↦ b}
```

Fig. 4. Redundant resources from all program points to the end of `f()`, along with the needed resources used to calculate them. Needed resources are prefixed with `n` and redundant resources are prefixed with `r`.

Next, the analysis looks for needed resources in the second thread `f2()`, and matches them against redundant resources in the first thread `f1()`. In this case, the analysis identifies two distinct needed resources, nx and ny.

$$\mathsf{nx}\colon x \mapsto v_1' \qquad\qquad \mathsf{ny}\colon y \mapsto v_2'$$

nx becomes needed in `f2()` at line 3. It becomes redundant in `f1()` at lines 6 and 13 of Figure 4. ny becomes needed at lines 6 and 13. It becomes redundant at lines 8 and 11.

The final stage of the analysis is to inject synchronization corresponding to these discovered resources. Specifically, `grant`-barriers are injected at the points that the resources become redundant, while `wait`-barriers are injected at the points the resources become needed. Synthesizing barriers follows a four-step process.

(1) Identify a needed resource r at some point in the later segment. This point may be chosen with assistance from the programmer, or heuristically, for example by favoring bigger needed resources – see Section 7.2 for a discussion.
(2) Down all other control-flow paths in the later segment, find a program point such that the needed resources is covered by r. This step ensures that the identified resources r is received down all possible paths in the program.
(3) Down all control-flow paths in the earlier segment, find a program point such that the redundant resource covers r. This step ensures that the resource r is satisfied down all possible paths in the program.

(4) Synthesize a channel name, and insert `wait()` into the later segment, and `grant()` into the earlier segment at the identified program points.

After each iteration of this process, the needed and redundant resources are updated to reflect the new synchronization. The analysis halts when no needed resources remain.

The parallelized version of `f()` is shown in Figure 1. The needed resources associated with barriers are selected heuristically. The analysis could also safely choose a single needed resource as follows.

$$x \mapsto v_1' * y \mapsto v_2'$$

This resource overapproximates both nx and ny. However, this resource would need to be acquired at a dominating program point for nx and ny, which would sequentialize the program by forcing the `wait`-barrier to the start of the second thread.

As there are only two heap locations, it is trivial to see that nx and ny are a sensible choice of needed resource. Choosing needed resources becomes more tricky when resources are dynamic structures, such as linked lists (Section 3.2).

3. GENERALIZING THE ANALYSIS

In the previous section we informally presented the core of our analysis. In this section, we discuss how it can be generalized to handle (a) an unbounded number of threads, and (b) dynamic (heap-allocated) data structures.

3.1. Unbounded Iterations: Parallel-For

Until this point we have assumed that the programmer identifies a fixed number of static program segments for parallelization—in the example, the two calls to `f()`. However, our analysis can support unbounded segments running in parallel. To do this, we introduce the new construct `pfor` (*parallel-for*). The intended semantics of a `pfor` is identical to a standard `for`-loop, but in which all iterations may run in parallel, with the analysis injecting synchronization as required.

For example, we might have the following program.

```
pfor(i=1; i++; i<n) {
  f(i);
}
```

The externally visible input-output behavior will be the same as the sequential composition.

```
f(1); f(2); f(3); ... f(n);
```

Internally, however, each call to `f()` runs concurrently.

Parallelization. Figure 5 shows the parallelization of the example `pfor`. A `pfor` is translated to a sequential `for` loop, in which each iteration forks a new copy of the parallelized loop body. Resources are transferred between the threads in order of thread-creation. That is, the nth iteration of the `pfor` acquires resources from the logically preceding $(n-1)$th iteration, and releases resources to the logically $(n+1)$th iteration.

This ordering is implemented through *shared channels*; a thread shares with its predecessor and successor a set of channels for receiving (i.e., waiting) and sending (i.e., signaling) resources, resp. The number of threads—and so, the required number of channels—is potentially decided at run-time. Consequently, channels are dynamically allocated in the main `for`-loop using the `newchan()` operation [Dodds et al. 2011]. Each iteration creates a set of new channels, and passes the prior and new set to the forked thread.

```
chan wx = newchan();
chan wy = newchan();
chan wx', wy';
grant(wx);
grant(wy);
for (i=0; i++; i<n) {
  wx' = newchan();
  wy' = newchan();
  fork(f(i,wx,wy,wx',wy'));
  wx = wx'; wy = wy';
}
wait(wx);
wait(wy);
```

```
f(i, wxp, wyp, wx, wy){
  wait(wxp);
  int v = *x;
  if (v >= i) {
    grant(wx);
    wait(wyp);
    *y = v;
    grant(wy);
  }
  else {
    wait(wyp);
    grant(wy);
    *x = 0;
    grant(wx);
  }
}
```

Fig. 5. Parallelization of our `f()`, generalized to deal with n threads.

```
move(s, l) {
  i = 0;
  x = s.nxt;
  p = s;
  while (i<l && x.nxt!=null) {
    i++;
    p = x;
    x = x.nxt;
  }
  while (x.nxt!=null)
    x = x.nxt;
  if (p.nxt!=x) {
    tmp = p.nxt.nxt;
    p.nxt.nxt = null;
    x.nxt = p.nxt;
    p.nxt = tmp;
  }
}
```

Fig. 6. Example function `move()`.

The parallelized version of `f()` (Figure 5) takes four channel arguments, a pair of each for `x` and `y`. The prior channels are used for resource transfer with the logically preceding thread (here `wx`, `wy`), and the new channels are used to communicate resource transfer with the logically following thread (`wx'`, `wy'`). As each iteration calls both `wait()` and `grant()`, to receive and send resources, the synchronization in Figure 5 merges the synchronization from both the first and second threads in Figure 1.

3.2. Loops and Dynamic Data Structures

Up to this point, we have dealt with channels transferring single heap locations injected into straight-line code. Our analysis can in fact handle more complicated heap-allocated data structures such as linked lists, and control-flow constructs such as loops. To illustrate, consider the function `move()`, shown in Figure 6. This function searches a linked list for the node at a particular position in the list (controlled by the parameter `l`) and moves that node to the end of the list.

$\{\mathsf{node}(\mathtt{s}, n_1') * \mathsf{node}(n_1', n_2') * \mathsf{lseg}(n_2', \mathsf{null})\}$
```
move(s, l) {
  i = 0;
  x = s.nxt;
  p = s;
```
$\{\mathsf{node}(\mathtt{s}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null}) \wedge \mathtt{s} = \mathtt{p}\}$
```
  while (i<l && x.nxt!=null) {
```
$\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null})\}$
```
    i++;
    p = x;
    x = x.nxt;
  }
```
$\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null})\}$
```
  while (x.nxt!=null) {
```
$\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, n_1') * \mathsf{lseg}(n_1', \mathtt{x}) * \mathsf{node}(\mathtt{x}, n_2') * \mathsf{lseg}(n_2', \mathsf{null})\}$
```
    x = x.nxt;
  }
```
$\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, n') * \mathsf{lseg}(n', \mathtt{x}) * \mathsf{node}(\mathtt{x}, \mathsf{null})\}$
```
  if (p.nxt!=x) {
```
$\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, n_1') * \mathsf{node}(n_1', n_2') * \mathsf{lseg}(n_2', \mathtt{x}) * \mathsf{node}(\mathtt{x}, \mathsf{null})\}$
```
    tmp = p.nxt.nxt;
    p.nxt.nxt = null;
    x.nxt = p.nxt;
    p.nxt = tmp;
```
$\{\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{tmp}) * \mathsf{lseg}(\mathtt{tmp}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{node}(n', \mathsf{null})\}$
```
  }
}
```
$\{\mathsf{node}(\mathtt{s}, n_1') * \mathsf{node}(n_1', n_2') * \mathsf{lseg}(n_2', \mathsf{null})\}$

Fig. 7. Separation logic proof of `move()`, a list-manipulating program whose automated parallelization requires reasoning over complex assertions and predicates.

We assume that the following specification holds:

$$\{\mathsf{node}(\mathtt{s}, n_1') * \mathsf{node}(n_1', n_2') * \mathsf{lseg}(n_2', \mathsf{null})\}$$
$$\texttt{move(s, l)}$$
$$\{\mathsf{node}(\mathtt{s}, n_3') * \mathsf{node}(n_3', n_4') * \mathsf{lseg}(n_4', \mathsf{null})\}$$

This specification is simple: all it says is that executing `move()` on a list of length two or more results in a list of length two or more. A proof of this specification is given in Figure 7.

An important feature of our approach is that the input safety proof need not specify all the relevant sequential properties; all sequential dependencies are enforced. Thus we can view this program as representative of a class of algorithms and implementations that traverse the head of a list, then mutate the tail. With minor modifications, the same proof pattern would cover assigning to the values in the list, or sorting the tail of the list.

We consider the parallelization of a sequential program consisting of pair of calls to the `move()`.

```
move(s,a); move(s,b);
```

```
move1(s, l) {
  i = 0;
  x = s.nxt;
  p = s;
  while (i<l && x.nxt!=null) {
    i++;
    p = x;
    x = x.nxt;
  }
  *pr = p;
  grant(i1);
  while (x.nxt!=null)
    x = x.nxt;
  if (p.nxt!=x) {
    ... // omitted
  }
  grant(i2);
}
```

```
move2(s, l) {
  i = 0;
  wait(i1);
  if (s==*pr || s.nxt==*pr) wait(i2);
  x = s.nxt;
  p = s;
  while (i<l && x.nxt!=null) {
    if (x.nxt==*pr) wait(i2);
    i++;
    p = x;
    x = x.nxt;
  }
  while (x.nxt!=null) {
    if (x.nxt==*pr) wait(i2);
    x = x.nxt;
  }
  if (p.nxt!=x) {
    ... // omitted
  }
}
```

Fig. 8. Parallelization of `move()` for two threads.

The parallelized version of this program consists of two parallel calls to modified versions of `move`.

```
move1(s,a) || move2(s,b)
```

The `wait()` and `grant()` barriers in $\texttt{move}_1\texttt{()}$ and $\texttt{move}_2\texttt{()}$ must enforce the following properties.

— $\texttt{move}_2\texttt{()}$ must not write to a list node until $\texttt{move}_1\texttt{()}$ has finished both reading from and writing to it. Consequently, $\texttt{move}_2\texttt{()}$ must wait for $\texttt{move}_1\texttt{()}$ to finish traversing a segment of the list before moving a node inside the segment.
— $\texttt{move}_2\texttt{()}$ must not read from a list node until $\texttt{move}_1\texttt{()}$ has finished writing to it. Consequently, $\texttt{move}_2\texttt{()}$ must wait for $\texttt{move}_1\texttt{()}$ to finish moving a node before it traverses over that point in the list.

This example is substantially more subtle to analyse and parallelize than our earlier one, because the list is not divided into segments that can be discovered from the static structure of the program. (A simple points-to analysis would be insufficient to discover the salient partitioning, for example.) In the worst case, a `wait()` at the start of $\texttt{move}_2\texttt{()}$ and a `grant()` at the end of $\texttt{move}_1\texttt{()}$ enforces sequential order. However, by reasoning about the structure of the manipulated list using the safety proof given in Figure 7, our approach can do considerably better.

The parallelized program synthesized by our algorithm is shown in Figure 8. This parallelization divides the list into two segments, consisting of the portions read and modified by $\texttt{move}_1\texttt{()}$. A shared heap-location `pr`, introduced by our algorithm, stores the address of the starting node of the portion modified by $\texttt{move}_1\texttt{()}$. The thread $\texttt{move}_2\texttt{()}$ uses `pr` to synchronize access to the second segment of the list.

Handling dynamic structures means dealing with allocation and disposal. Fortunately, separation logic handles both straightforwardly. Updates to the data structure and object allocation are by assumption reflected in the invariants of the original sequential proof. Thus updates and allocations are also reflected in the invariants which our analysis constructs to represent the contents of channels. However,

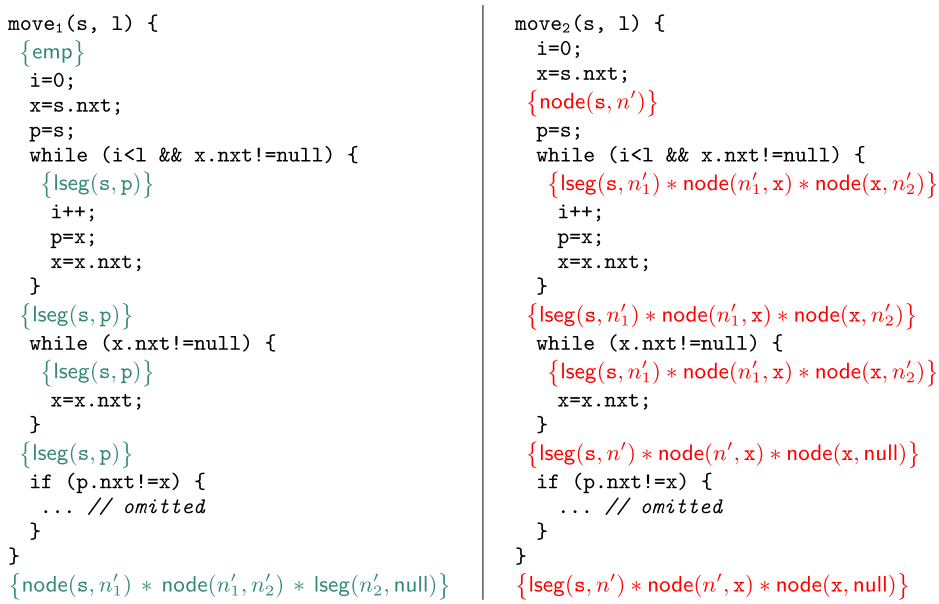


Fig. 9. Redundant resources (from the start to other program points) and needed resources (from other program points to the end) for `move()`.

introducing allocation and disposal affects the behavior-preservation result discussed in Section 6.

Handling list segments. We run our resource-usage analysis over the program to determine redundant and needed resources. The results of the analysis are shown in Figure 9.

Consider the point in Figure 9 just before the start of the second while loop. Our analysis discovers that only the resource $\mathsf{lseg}(\mathtt{p}, \mathsf{null})$ is needed to execute from the start of this loop to the end of the function. Comparing this resource to the corresponding invariant in the sequential proof reveals that the resource $\mathsf{lseg}(\mathtt{s}, \mathtt{p})$ is redundant at this point. This assertion represents the segment of the list that has already been traversed by $\mathtt{move}_1()$, from the head of the list to `p`.

Injecting barriers. `grant()` and `wait()` barriers are injected into `move()` as discussed above, in Section 2.6. Barriers `wait()` and `grant()` should only be called once for any given channel in the program. For simplicity of exposition, in Figure 8, we write $\mathtt{move}_2()$ with `wait()` controlled by conditionals. However, in general this approach requires us to modify loop invariants, which is often difficult. In the formal definition of the analysis, we only inject barriers into loop-free code, and perform syntactic loop-splitting to expose points in loops where barriers can be called conditionally. Details are given in Section 5.2.

Value materialization. In order to safely transfer a redundant resource to logically later program segments we need the assertion to be expressed in global variables, and to be invariant from the point in time that the resource is released, to the point it is received in the subsequent thread. The assertion $\mathsf{lseg}(\mathtt{s}, \mathtt{p})$ initially generated by the analysis satisfies neither condition, because it is partly expressed in terms of the local variable `p`, which changes during execution of $\mathtt{move}_1()$. The current thread may invalidate it by changing the value stored in `x`.

To satisfy this requirement, we could simply existentially quantify the offending variable, p, giving the redundant resource

$$\exists y.\, \mathsf{lseg}(\mathtt{s}, y).$$

However, such a weakening loses important information, in particular the relationship between the resource, and the list tail beginning at p. To retain such dependency relationships, our analysis stores the current value of p into a global location pr shared between $\mathtt{move}_1()$ and $\mathtt{move}_2()$. (We call storing a snapshot of a local value in this way *materialization*). An assignment is injected into $\mathtt{move}_1()$ at the start of the second loop.

```
...
*pr = p;
while (x.nxt!=null)
...
```

After the assignment, the proof needs to be modified to accommodate the new global location pr. In this case, this amounts to simply modifying the invariant to add an extra points-to assertion representing the new location:

$$\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null}) * \mathtt{pr} \mapsto \mathtt{p}.$$

The redundant state can now be described as follows.

$$\mathsf{lseg}(\mathtt{s}, y') * \mathtt{pr} \overset{1/2}{\longmapsto} y'$$

The assertion $\mathtt{pr} \overset{1/2}{\longmapsto} y'$ represents fractional, read-only permission on the shared location pr. This binds together the head of the list and the remainder of the list when they are recombined, and thus helps preserve the list invariant.

When traversing the list, $\mathtt{move}_2()$ compares its current position with pr. If it reaches the pointer stored in pr, it must wait to receive the second, remainder segment of the list from $\mathtt{move}_1()$.

4. TECHNICAL BACKGROUND

In Section 2 and Section 3 we gave an overview of our analysis. We now turn to a formal description, which we start with a technical background about the programming language, the assertion language and the proof representation.

4.1. Programming Language and Representation

We assume the following heap-manipulating language.

$$\begin{aligned}
e &::= \mathtt{x} \mid \mathtt{nil} \mid t(\bar{e}) \mid \ldots && \textit{(expressions)}\\
b &::= \mathtt{true} \mid \mathtt{false} \mid e = e \mid e \neq e \mid \ldots && \textit{(booleans)}\\
a &::= \mathtt{x} := e \mid \mathtt{x} := *e \mid *\mathtt{x} := e \mid \mathtt{x} := \mathtt{alloc}() \mid \ldots && \textit{(primitive commands)}\\
C &::= C; C \mid \ell\colon \mathtt{skip} \mid \ell\colon a \mid \ell\colon \mathtt{if}(b)\,\{C\}\,\mathtt{else}\,\{C\} \mid \ell\colon \mathtt{while}(b)\,\{C\}
\end{aligned}$$

To avoid an extra construct, we define $\mathtt{for}(C_1; C_2; b)\{C_3\}$ as $C_1; \mathtt{while}(b)\{C_2; C_3\}$.

```
void f(i) {
ℓ1: int v = *x;
ℓ2: if (v >= i) {
ℓ3: *y = v;
    }
    else {
ℓ4: *x = 0;
    }
}
```

$$
\begin{array}{rl}
\mathtt{f_s}\colon & \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_1\colon & \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2\colon & \mathtt{v} = a \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2\ell_3\colon & \mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2(\ell_2)^{\mathtt{t}}_{\mathbf{e}}\colon & \mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto a \\
\ell_2\ell_4\colon & \mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b \\
\ell_2(\ell_4)^{\mathtt{f}}_{\mathbf{e}}\colon & \mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto b \\
\mathtt{f_e}\colon & (a \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto a) \vee (a < \mathtt{i} \wedge \mathtt{x} \mapsto 0 * \mathtt{y} \mapsto b)
\end{array}
$$

Fig. 10. Left: labels for commands in function `f()` from §2. Right: associated assertions in the sequential proof of `f()`.

We define our algorithm for the multi-iteration construct `pfor` defined in Section 3.1. To simplify the exposition, we assume a fixed input program $\mathbb{P}$ of the following form.

```
global r̄;

work(x̄){
  local ȳ;
  C;
}

main(void){
  pfor(C1; C2; b) {
    work();
  }
}
```

Parallelization generates a new program $\mathbb{P}_{par}$ that executes a transformed version of `work` in separate threads.

Labeling of commands. We assume that every command $C \in \mathsf{Cmd}$ in the program is indexed by a unique label $\ell \in \mathsf{Label}$; the function identifier `work` is also treated as a label. We identify a particular command by its label, and when needed denote by $\mathsf{cmd}(\ell)$ the command at the label ℓ. Functions $\mathsf{pred}, \mathsf{succ}\colon \mathsf{Label} \rightharpoonup \mathsf{Label}$ return the label of the previous (the next, resp.) command in a block. For a label ℓ corresponding to a `while` loop, the predecessor of the first command and successor of the last command in the block are denoted by $\ell_{\mathbf{s}}$ and $\ell_{\mathbf{e}}$, respectively. For ℓ corresponding to `if-else`, $\ell^{\mathtt{t}}_{\mathbf{s}}$ ($\ell^{\mathtt{t}}_{\mathbf{e}}$) and $\ell^{\mathtt{f}}_{\mathbf{s}}$ ($\ell^{\mathtt{f}}_{\mathbf{e}}$) are labels of the predecessor of the first (the successor of the last) commands in the `if` and `else` branches, respectively. The left-hand side of Figure 10 shows a labeling of the function `f()` from Section 2.

Program representation. The program's `work` function is represented by a variant of abstract syntax tree (formally, an ordered forest) $\mathcal{T}$ with $\mathsf{Label} \cup \{\mathtt{work}_{\mathbf{s}}, \mathtt{work}_{\mathbf{e}}\}$ as the set of nodes and as the set of edges all pairs (ℓ, ℓ') where ℓ is a label of an `if-else` block and ℓ' a label of a command within the encompassed `if` or `else` branches. Labels ℓ and ℓ' that belong to the same block are siblings in the tree. They are ordered $\ell < \ell'$ if there exists $n \geq 1$ such that $\ell' = \mathsf{succ}^n(\ell)$. We denote by $\ell_{\uparrow}$ and $\ell_{\downarrow}$ the smallest (resp. largest) label in the block containing ℓ. We write $[\ell, \ell'\rangle$ to denote the sequence of labels between ℓ inclusively and ℓ' exclusively, and $\mathbb{P}_{[\ell,\ell'\rangle}$ for the corresponding program fragment from ℓ (inclusively) to ℓ' (exclusively).

Program paths. A *program path* is a finite sequence of nodes in the tree $\mathcal{T}$. Each program path determines a sequence of conditionals that need to be traversed in order to reach a particular point in the program. We denote the set of all program paths by Paths.

We often want to manipulate and compare program paths. For $\gamma = \ell_1 \dots \ell_n \in \mathsf{Paths}$, we write $\gamma[i]$ to denote ℓ_i, $\gamma[i..j]$ to denote $\ell_i \dots \ell_j$ and $|\gamma|$ to denote the length n. For program paths γ and γ', $\gamma \curlywedge \gamma'$ denotes their longest common prefix, that is, for $k =$

$|\gamma \curlywedge \gamma'|$ we have $\forall j \leq k, \gamma \curlywedge \gamma' = \gamma[j] = \gamma'[j]$ and if $|\gamma|, |\gamma'| > k$ then $\gamma[k+1] \neq \gamma'[k+1]$. We define a partial order $\prec$ on Paths as follows. We say that $\gamma \prec \gamma'$ iff $\gamma \curlywedge \gamma' = \gamma$ and $|\gamma| < |\gamma'|$, or for $k = |\gamma \curlywedge \gamma'|$ we have $|\gamma|, |\gamma'| > k$ and $\gamma[k+1] < \gamma'[k+1]$. (Intuitively, two paths are related in this way if they share a prefix, and one takes predecessor branch to the other in the same block). We say that $\gamma \preceq \gamma'$ iff $\gamma \prec \gamma'$ or $\gamma = \gamma'$.

LEMMA 4.1. (Paths, $\preceq$) *is a lattice with least element* $\mathtt{work_s}$ *and greatest element* $\mathtt{work_e}$.

For $\Gamma \subseteq$ Paths we define $\max(\Gamma)$ as $\max(\Gamma) \triangleq \{\gamma \in \Gamma \mid \neg\exists \gamma' \in \Gamma \,.\, \gamma \prec \gamma'\}$. We define $\min(\Gamma)$ analogously. For $\Gamma, \Gamma' \subseteq$ Paths we write $\Gamma \preceq \Gamma'$ if for all $\gamma \in \Gamma$, there exists $\gamma' \in \Gamma'$ such that $\gamma \preceq \gamma'$.

4.2. Assertion Language

We assume that the assertions in the program proof are expressed using a class of separation logic formulae called *symbolic heaps*. A symbolic heap Δ is a formula of the form $\exists \bar{x} \,.\, \Pi \wedge \Sigma$ where Π (the pure part) and Σ (the spatial part) are defined by the following.

$$\Pi ::= \mathsf{true} \mid e = e \mid e \neq e \mid p(\bar{e}) \mid \Pi \wedge \Pi$$
$$\Sigma ::= \mathsf{emp} \mid s(\bar{e}) \mid \Sigma * \Sigma$$

Here $\bar{x}$ are logical variables, e ranges over expressions, $p(\bar{e})$ is a family of pure (first-order) predicates (such as, e.g., arithmetic inequalities, etc), and $s(\bar{e})$ a family of spatial predicates (such as, e.g., points-to $x \mapsto e$, singly linked list $\mathsf{lseg}(e, e)$, doubly linked lists, trees, etc). We refer to the pure and the spatial part of an assertion Δ as Δ^Π and Δ^Σ respectively. We extend Π with the $\vee$ connective to give the set of quantifier-free first-order formulae $\Pi_\vee$.

We often need to substitute variables, for example when recasting an assertion into a different calling context. If $\varrho = \bar{x} \mapsto \bar{e}$ is a mapping from variables in $\bar{x}$ to $\bar{e}$ then $\Delta[\varrho]$ denotes the formula obtained by substituting every occurrence of x_i in Δ with the corresponding e_i. We denote by δ^{-1} the inverse variable mapping, if δ is injective[2].

We treat a disjunction of symbolic heaps as a set and interchangeably use the $\cup$ and $\vee$ operators. Such disjunctions will often arise in assertions at join points in the program. The set of all symbolic heaps is denoted by SH and the set of all disjunctive symbolic heaps by $\mathcal{P}(\mathsf{SH})$. We overload the $\wedge$ and $*$ operators in a natural way: for $\Delta_i = \Pi_i \wedge \Sigma_i$, $i = 1, 2$, we define the following.

$$\Delta_1 * \Delta_2 \triangleq (\Pi_1 \wedge \Pi_2) \wedge (\Sigma_1 * \Sigma_2)$$
$$\Pi \wedge \Delta_i \triangleq (\Pi \wedge \Pi_i) \wedge \Sigma_i$$
$$\Sigma * \Delta_i \triangleq \Pi_i \wedge (\Sigma * \Sigma_i)$$

Operators $\wedge$ and $*$ distribute over $\vee$, thus we allow these operations on disjunctive heaps just as if they were on symbolic heaps and furthermore use the same notation Δ to refer to both symbolic and disjunctive symbolic heaps.

[2]In our framework, substitutions are always guaranteed to be injective because the variables being substituted correspond to heap locations and channel resources whose denotations are guaranteed to be distinct; if the substitution involves values, then they must be implicitly existentially quantified, and can therefore be assumed to be distinct.

4.3. Theorem Prover

Our resource-usage analysis relies on a sufficiently powerful (automated) prover for separation logic that can deal with three types of inference queries.

— $\Delta_1 \vdash \Delta_2 * [\Delta_F]$ (frame inference): given Δ_1 and Δ_2 find the frame Δ_F such that $\Delta_1 \vdash \Delta_2 * \Delta_F$ holds;
— $\Delta_1 * [\Delta_A] \vdash \Delta_2$ (abduction): given Δ_1 and Δ_2 find the "missing" assumption Δ_A such that $\Delta_1 * \Delta_A \vdash \Delta_2$ holds;
— $\Delta_1 * [\Delta_A] \vdash \Delta_2 * [\Delta_F]$ (bi-abduction): given Δ_1 and Δ_2 find Δ_A and Δ_F such that $\Delta_1 * \Delta_A \vdash \Delta_2 * \Delta_F$ holds.

As before, square brackets denote the portion of the entailment to be computed. We sometimes write $[\_]$ for a portion that should be computed but that is existentially quantified and will not be reused.

None of these queries has a unique answer in general. However, for the soundness of our resource-usage analysis any answer is acceptable (though some will give rise to a better parallelization than the others). Existing separation logic tools generally supply only a single answer.

We note that the theorem prover support for these inference queries is required independently of how the program proof has been obtained (constructed by a program analysis tool or provided manually by a programmer). That is, our resource-usage analysis is crucially dependent on the underlying theorem prover but not on the program analysis (or manual effort) that constructed the original sequential proof.

4.4. Sequential Proof

We assume a separation logic proof of the function `work`, represented as a map $\mathfrak{P} : \mathsf{Label} \to \mathcal{P}(\mathsf{SH})$.[3] Assertions in the proof have the property that for any label ℓ executing the program from a state satisfying $\mathfrak{P}(\ell)$ up to some subsequent label ℓ' will result in a state satisfying $\mathfrak{P}(\ell')$, and will not fault. Our approach is agnostic to the method by which the proof is created: it can be discovered automatically (e.g., by a tool such as Abductor [Calcagno et al. 2011]), prepared by a proof assistant, or written manually.

We assume that the structure of the proof is *modular*, that is, that primitive commands and loops at each label use specifications which are given a priori "in isolation", inherent only to the underlying programming language and agnostic to the particular context in which the specification is used. (This property is a standard feature of separation logic). We assume functions $\mathsf{Pre}, \mathsf{Post} : \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ associating labels of primitive commands with such modular pre- and postconditions, respectively. That is, the command at label ℓ has a specification

$$\{\mathsf{Pre}(\ell)\}\ \mathsf{cmd}(\ell)\ \{\mathsf{Post}(\ell)\},$$

which is applied in the actual proof by using the frame rule and appropriate variable substitutions. We also assume a function $\mathsf{Inv} : \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ associating `while` labels with loop invariants, which are used similarly as specifications.

Specifications and loop invariants are formally applied in the proof by using a mapping Ω from labels to variable substitutions such that $\Omega(\ell) = \bar{x} \mapsto \bar{x'}$ maps formal variables $\bar{x}$ in the specification assertion to actual variables $\bar{x'}$ in the proof assertion at a label ℓ. We write $\Delta[\Omega(l)]$ to represent the heap constructed by applying the substitutions defined by $\Omega(l)$ to the assertion Δ.

[3]We use the disjunctive (the powerset) completion of symbolic heaps as the domain of proof assertions to signify use of the logical $\vee$ as the join operation. We made this design choice to the sake of simplicity, however, more elaborate join operations (e.g., Yang et al. [2008]) could be accommodated.

We also assume a function $\mathfrak{F} : \mathsf{Label} \rightharpoonup \mathcal{P}(\mathsf{SH})$ returning the framed portion of the assertion at a particular program label. Then at each label ℓ we have $\mathfrak{P}(\ell) \vdash \mathsf{Pre}(\ell)[\Omega(\ell)] * \mathfrak{F}(\ell)$ and $\mathsf{Post}(\ell)[\Omega(\ell)] * \mathfrak{F}(\ell) \vdash \mathfrak{P}(\mathsf{succ}(\ell))$. If ℓ is a `while` label then $\mathsf{Pre}(\ell)$ and $\mathsf{Post}(\ell)$ are replaced by a single $\mathsf{Inv}(\ell)$.

We extend functions $\mathfrak{P}$ and $\mathfrak{F}$ from labels to program paths by taking the assertion associated with the last label. That is, if $\gamma = \ell_1 \dots \ell_n$ then we define $\mathfrak{P}(\gamma) \triangleq \mathfrak{P}(\ell_n)$ and $\mathfrak{F}(\gamma) \triangleq \mathfrak{F}(\ell_n)$. The right-hand side of Figure 10 shows a proof for the function `f()` from Section 2 laid out with respect to the associated program paths.

5. PARALLELIZATION ALGORITHM

We now formally define our parallelization algorithm. The goal of our approach is to construct a parallelized version of the input program $\mathbb{P}$. In particular, our approach generates a new function `work`$'$ in $\mathbb{P}_{par}$ as follows.

```
main_{P_par}(void) {
  // channel initialization.
  for(C_1; C_2; b) {
    // channel creation.
    fork(work', ...);
  }
  // channel finalization.
}
```

As described in the intuitive development in Section 2.4, our parallelization algorithm is composed of a resource usage analysis (Section 5.1) and parallelizing transformation (Section 5.2). The parallelizing transformation should be viewed as just one application of the resource usage analysis; more optimized proof-preserving parallelizing transformations are certainly possible. Our overall aim for this section is to present a framework for resource-sensitive dependency-preserving analyses.

5.1. Resource Usage Analysis

The resource usage analysis computes two maps:

$$\mathsf{needed}\colon \mathsf{Paths} \times \mathsf{Paths} \rightharpoonup \mathcal{P}(\mathsf{SH}),$$
$$\mathsf{redundant}\colon \mathsf{Paths} \times \mathsf{Paths} \rightharpoonup \mathcal{P}(\mathsf{SH}).$$

For $\gamma_s, \gamma_e \in \mathsf{Paths}$, such that $\gamma_s \prec \gamma_e$, $\mathsf{needed}(\gamma_s, \gamma_e)$ gives the resources that might be accessed during execution from γ_s to γ_e. In parallelization, these are the resources that must be acquired before execution of the current thread can proceed. Similarly, $\mathsf{redundant}(\gamma_s, \gamma_e)$ gives the resources that are guaranteed not to be accessed by the program between γ_s and γ_e. In parallelization, these are the resources that can safely be transferred to other parallel threads.

We calculate needed resources in a program by using two functions: `Needed-Block`, which computes needed resources between labels within the same program block, and `Needed`, which computes needed resources between arbitrary program paths, spanning possibly different blocks. Intuitively, the needed resources computation can be seen as a *liveness* analysis for resources—a resource R which is needed between program paths γ_s and γ_e is *live* in the sense that on the way from γ_s to γ_e some command might access R by virtue of command's specification mentioning R.

The function `Needed-Block` (Algorithm 1) uses backward symbolic execution to compute needed resources between a pair of labels (ℓ', ℓ'') located within the same program block. The execution process starts by an arbitrary symbolic heap Δ, representing the

ALGORITHM 1: Computing needed resources locally using backward symbolic execution.

```
1  Needed-Block((ℓ′, ℓ″): Label × Label, Δ: 𝒫(SH))
2  begin
3      ℓ := ℓ″;
4      while ℓ ≠ ℓ′ do
5          ℓ := pred(ℓ);
6          if cmd(ℓ) matches if(b){C}else{C′} then
7              Δ := 𝔓(ℓ)^Π ∧ (Needed-Block((ℓ^t_s, ℓ^t_e), Δ) ∪ Needed-Block((ℓ^f_s, ℓ^f_e), Δ));
8          end
9          if cmd(ℓ) matches while(b){C} then
10             Inv(ℓ)[Ω(ℓ)] ∗ [Δ_A] ⊢ Δ ∗ [_];
11             Δ := 𝔓(ℓ)^Π ∧ (Inv(ℓ)[Ω(ℓ)] ∗ Δ_A);
12         end
13         else
14             Post(ℓ)[Ω(ℓ)] ∗ [Δ_A] ⊢ Δ ∗ [_];
15             Δ := 𝔓(ℓ)^Π ∧ (Pre(ℓ)[Ω(ℓ)] ∗ Δ_A);
16         end
17     end
18     return Δ;
19 end
```

resources we have at the end of the block. By pushing Δ backwards along the preceding commands, `Needed-Block` successively discovers missing resources that are needed to execute the commands in the block. The algorithm poses bi-abduction queries to determine the sufficient precondition of primitive commands and loops (using the loop invariants as specifications) and dives recursively into the blocks of `if` and `else` branches. The existing pure components of the sequential proof are used to strengthen the abduced solutions returned by the prover. The use of pure assertions from the sequential proof is essential in practice—abducing missing pure assertions is much harder than abducing spatial resources and would likely fail often. Upon completion, `Needed-Block` returns a symbolic heap sufficient to execute the program block $\mathbb{P}_{[\ell',\ell'')}$.

The key step of the algorithm is performed on lines 14 and 15. These two steps "simulate" backward execution of the command at label ℓ, $\mathsf{cmd}(\ell)$, by using its specification. The goal of such process is to discover the eventually missing resources that $\mathsf{cmd}(\ell)$ needs for safe execution. Let us explain how these two steps work in more detail.

The command at label ℓ has a specification $\{\mathsf{Pre}(\ell)\}\ \mathsf{cmd}(\ell)\ \{\mathsf{Post}(\ell)\}$. Let Δ be the symbolic heap representing the needed resources for safe execution of commands below label ℓ that we have computed in previous iterations. For $\mathsf{cmd}(\ell)$ to have been executed safely, it must have been able to establish $\mathsf{Post}(\ell)$ upon completion. Since in general we do not know in which capacity Δ contains the resources described by $\mathsf{Post}(\ell)$, we pose a bi-abduction query

$$\mathsf{Post}(\ell)[\Omega(\ell)] * [\Delta_A] \vdash \Delta * [\_]\,,$$

forcing the post-state after execution of $\mathsf{cmd}(\ell)$ to become $\mathsf{Post}(\ell)[\Omega(\ell)] * \Delta_A$. By framing Δ_A away and using the specification of $\mathsf{cmd}(\ell)$, we obtain $\mathsf{Pre}(\ell)[\Omega(\ell)] * \Delta_A$ as the pre-state before the execution of $\mathsf{cmd}(\ell)$. Finally, we enrich the computed prestate with the pure assertion $\mathfrak{P}(\ell)^{\Pi}$ from the sequential proof and store the resulting symbolic heap as the new Δ.

LEMMA 5.1. *For every Δ, the symbolic heap returned by* `Needed-Block`$((\ell',\ell''),\Delta)$ *is a sufficient precondition for* $\mathbb{P}_{[\ell',\ell'')}$.

ALGORITHM 2: Computing needed resources between blocks.

```
1  Needed(γs : Paths, γe : Paths)
2  begin
3      k := |γe|;
4      Δ := 𝔓(γe)^Π ∧ emp;
5      while k > |γs ⅄ γe| + 1 do
6          Δ := Needed-Block((γe[k]↑, γe[k]), Δ);
7          k := k − 1;
8      end
9      Δ := Needed-Block((γe[k], γs[k]), Δ);
10     while k < |γs| do
11         k := k + 1;
12         Δ := Needed-Block((γs[k], γs[k]↓), Δ);
13     end
14     return Δ;
15 end
```

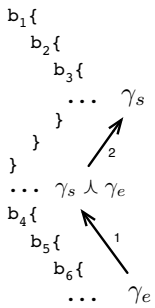
PROOF. Follows by iteratively applying the disjunctive version of the frame rule:

$$\frac{\{P\}\, C\, \{\bigvee_{i\in\mathcal{I}} Q_i\} \qquad \Delta * \bigvee_{j\in\mathcal{J}} \Delta_j^A \vdash P * \bigvee_{k\in\mathcal{K}} \Delta_k^F}{\{\bigvee_{j\in\mathcal{J}} (\Delta * \Delta_j^A)\}\, C\, \{\bigvee_{i\in\mathcal{I},k\in\mathcal{K}} (Q_i * \Delta_k^F)\}}$$

and Hoare's rule of composition. □

The function `Needed` (Algorithm 2) lifts `Needed-Block` from within blocks to between blocks and calculates sufficient precondition between arbitrary program paths. Given two assertion points represented as program paths γ_s and γ_e, `Needed`(γ_s, γ_e) works by successively pushing backwards the assertion $\mathfrak{P}(\gamma_e)^{\Pi} \wedge \mathsf{emp}$ from γ_e to γ_s. In the first while loop (line 5), the algorithm steps backwards from γ_e towards the nearest ancestor block containing both γ_s and γ_e, designated by their longest common prefix. After stepping through that block, in the second while loop (line 10) the algorithm keeps stepping backwards, proceeding inwards towards γ_s. Both phases of the algorithm use Algorithm 1 to compute the needed resources in-between program blocks.

The following diagram illustrates the behavior of Algorithm 2:



Arrow 1 corresponds to the first while loop in the algorithm (line 5), while arrow 2 corresponds to the second while loop (line 10).

We tabulate the results computed by `Needed()` into a map needed as follows:

— if $\mathfrak{P}(\gamma_s) \vdash$ `Needed`$(\gamma_s, \gamma_e) * [\_]$, then $\mathsf{needed}(\gamma_s, \gamma_e) := \mathfrak{P}(\gamma_s)^{\Pi} \wedge$ `Needed`(γ_s, γ_e);
— otherwise, $\mathsf{needed}(\gamma_s, \gamma_e) := \mathfrak{P}(\gamma_s)$.

ALGORITHM 3: Computing redundant resources

```
1 Redundant(γs : Paths, γe : Paths)
2 begin
3 |  𝔓(γs) ⊢ Needed(γs, γe) ∗ [ΔR];
4 |  return ΔR;
5 end
```

	ℓ_2	$\ell_2\ell_3$	$\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}$	$\ell_2\ell_4$	$\ell_2(\ell_2)^{\mathtt{f}}_{\mathtt{e}}$	$\mathtt{f_e}$
ℓ_1	$\mathtt{x} \mapsto a$	$a \geq \mathtt{i} \wedge$ $\mathtt{x} \mapsto a$	$a \geq \mathtt{i} \wedge$ $\mathtt{x} \mapsto a * \mathtt{y} \mapsto b$	$a < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a$	$a < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a$	$(a \geq \mathtt{i} \wedge$ $\mathtt{x} \mapsto a * \mathtt{y} \mapsto b)$ $\vee (a < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a)$
ℓ_2		$\mathtt{v} = a \wedge$ $\mathtt{v} \geq \mathtt{i}$	$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge$ $\mathtt{y} \mapsto b$	$\mathtt{v} = a \wedge$ $\mathtt{v} < \mathtt{i}$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a$	$(\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge$ $\mathtt{y} \mapsto b)$ $\vee (\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a)$
$\ell_2\ell_3$			$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge$ $\mathtt{y} \mapsto b$			$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge$ $\mathtt{y} \mapsto b$
$\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}$						$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i}$
$\ell_2\ell_4$					$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge$ $\mathtt{x} \mapsto a$
$\ell_2(\ell_2)^{\mathtt{f}}_{\mathtt{e}}$						$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i}$

Fig. 11. The needed map for the function `f()` from Section 2 (some entries omitted).

LEMMA 5.2. $\mathsf{needed}(\gamma_s, \gamma_e)$ *is a sufficient precondition to execute* $\mathbb{P}$ *from* γ_s *to* γ_e.

For further steps of the parallelization algorithm, we must ensure that the needed map is monotonic with respect to program paths, that is, that $\forall \gamma, \gamma', \gamma'' \in \mathsf{Paths}$ such that $\gamma \prec \gamma' \prec \gamma''$ we have $\mathsf{needed}(\gamma, \gamma'') \vdash \mathsf{needed}(\gamma, \gamma') * [\,]$. This ensures that the resources needed to reach a particular point include all the resources needed to reach preceding points. If the underlying theorem prover behaves consistently with respect to failing and precision then this property will hold. If that is not the case, we postprocess the map and insert additional assertions from the sequential proof as needed.

We now turn to computing redundant resources. The redundant resource between two program paths is a portion of the proof assertion in $\mathfrak{P}$ that is not required by the needed map. We calculate this portion by frame inference, as shown in function `Redundant` (Algorithm 3). We tabulate redundant resources in the map redundant by setting $\mathsf{redundant}(\gamma_s, \gamma_e) := \mathtt{Redundant}(\gamma_s, \gamma_e)$.

LEMMA 5.3. *If* $\mathfrak{P}(\gamma_s) \vdash \mathsf{redundant}(\gamma_s, \gamma_e) * [\Delta]$ *then* Δ *is a sufficient precondition to execute* $\mathbb{P}$ *from* γ_s *to* γ_e.

Consider the function `f()` from Section 2. The map obtained by `Needed` is shown in Figure 11. Figure 12 shows the map computed by `Redundant` with respect to the program path $\mathtt{f_e}$.

5.2. Parallelizing Transformation

We now describe the parallelizing transformation based on our resource-usage analysis. The construction proceeds in two phases. First, we compute program points for resource transfer and the associated resources that will be transferred between threads. Then we inject `grant` and `wait` barriers to realize this resource transfer. The resource transfer mechanism transfers a resource from one invocation of the `work` function to another in the parallelized version of the program.

	$\mathtt{f_e}$
ℓ_1	$a < \mathtt{i} \wedge \mathtt{y} \mapsto b$
ℓ_2	$(\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a) \vee (\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto b)$
$\ell_2\ell_3$	$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a$
$\ell_2(\ell_2)^{\mathtt{t}}_{\mathtt{e}}$	$\mathtt{v} = a \wedge \mathtt{v} \geq \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b$
$\ell_2\ell_4$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{y} \mapsto b$
$\ell_2(\ell_2)^{\mathtt{f}}_{\mathtt{e}}$	$\mathtt{v} = a \wedge \mathtt{v} < \mathtt{i} \wedge \mathtt{x} \mapsto a * \mathtt{y} \mapsto b$

Fig. 12. The redundant map with respect to the program path $\mathtt{f_e}$ for the function `f()` from Section 2.

Conditions on released and acquired resources. In the first phase we determine resources that should be released and acquired at particular points in the parallelized program. Released resources cannot be revoked, that is, each released resource should be included in the redundant map from the point of the release to the end of the `work` function—this way we know the resource will not be needed further. Acquired resources are held by the executing thread until released. Resources that are acquired along a sequence of program paths should contain what is prescribed by the needed map between each of the program paths.

We represent the result of this phase of the algorithm via the following maps:

— resource: ResId $\to \mathcal{P}$(SH), denoting resource identifiers that identify released and acquired resources selected by the algorithm;
— released: Paths $\rightharpoonup$ ResId $\times$ Subst, representing resources that are going to be released at a program path together with the variable substitution applied at that point;
— acquired: Paths $\rightharpoonup$ ResId, representing resources that are going to be acquired at a program path.

We require the following well-formedness properties of the maps.

(1) $\forall \gamma \in \mathrm{dom}(\mathsf{released})$.

$$\forall \gamma' \succ \gamma \,.\, \mathsf{released}(\gamma) = (r, \rho) \Rightarrow (\mathsf{redundant}(\gamma, \gamma') \vdash \mathsf{resource}(r)[\rho] * [\_]).$$

(2) $\forall \gamma \in \mathsf{Paths}$.

$$\circledast_{r \in \mathrm{dom}(\mathsf{resource})} \{\mathsf{resource}(r) \mid \exists \gamma' \prec \gamma \wedge \mathsf{acquired}(\gamma') = r\} \vdash \mathsf{needed}(\gamma, \mathtt{work_e}) * [\_].$$

(3) $\forall \gamma \in \mathrm{dom}(\mathsf{released})$.

$$\circledast_{r \in \mathrm{dom}(\mathsf{resource})} \{\mathsf{resource}(r) \mid \exists \gamma' \prec \gamma \wedge \mathsf{acquired}(\gamma') = r\} \vdash \mathsf{released}(\gamma) * [\_].$$

The first property states we can release only resources that are not needed between the given program path and any subsequent one. The second property states that the resources needed at a program path must have already been acquired. The third property states that only the resources that have been previously acquired can be released.[4]

In general, there are many solutions satisfying properties 1–3. For instance, there is always a trivial solution that acquires $\mathsf{needed}(\mathtt{work_s}, \mathtt{work_e})$ before the first command and releases it after the last, causing each invocation of `work` to be blocked until the preceding invocation finishes the last command. Of course, some solutions are better than others. Determining a solution's practical quality a priori would need to take into account the runtime behavior of synchronization, and that kind of analysis is beyond

[4]We could relax the third requirement if we extended our barriers to support *renunciation* [Dodds et al. 2011], the ability to release a resource without first acquiring it. Renunciation allows a resource to "skip" iterations, giving limited out-of-order signaling.

ALGORITHM 4: A heuristic algorithm for computing released and acquired resources.

1 $\mathsf{N} := \mathsf{needed}$; $\mathsf{R} := \mathsf{redundant}$;
2 $\mathcal{C} := \max\{\gamma \in \mathsf{Paths} \mid \mathsf{N}(\mathtt{work}_\mathbf{s}, \gamma) = \mathsf{emp}\}$;
3 **while** $\mathcal{C} \neq \{\mathtt{work}_\mathbf{e}\}$ **do**
4 $\quad \Sigma_r := \mathsf{N}(\mathit{choose}(\{(\gamma, \gamma') \mid \gamma \in \mathcal{C} \ \wedge \ \gamma' \in \mathsf{Paths} \ \wedge \ \gamma \prec \gamma'\}))^\Sigma$;
5 $\quad \mathcal{C}_\mathsf{N} := \{\gamma \in \mathcal{C} \mid \exists \gamma' \in \mathsf{Paths}.\, \mathsf{N}(\gamma, \gamma') \vdash \Sigma_r * [\_]\}$;
6 $\quad \varrho := \bar{\mathrm{x}} \mapsto \bar{\mathrm{x}}'$, where $\bar{\mathrm{x}}'$ fresh;
7 $\quad \Sigma'_r := \Sigma_r[\varrho]$;
8 $\quad \mathcal{C}_\mathsf{R} := \min\{\gamma' \in \mathsf{Paths} \mid \mathsf{R}(\gamma', \mathtt{work}_\mathbf{e})^\Sigma \vdash \Sigma'_r * [\_] \ \wedge \exists \gamma \in \mathcal{C}_\mathsf{N}.\, \gamma \preceq \gamma'\}$;
9 $\quad$ **if** $\mathcal{C}_\mathsf{N} \preceq \mathcal{C}_\mathsf{R}$ **then**
10 $\quad\quad r :=$ fresh resource id;
11 $\quad\quad \mathsf{resource}(r) := \Sigma'_r$;
12 $\quad\quad$ **forall the** $\gamma \in \mathcal{C}_\mathsf{R}$ **do**
13 $\quad\quad\quad \mathsf{released}(\gamma) := (r, \varrho)$;
14 $\quad\quad\quad$ **forall the** γ' *s.t.* $\gamma \preceq \gamma'$ **do**
15 $\quad\quad\quad\quad \mathsf{R}(\gamma', \mathtt{work}_\mathbf{e})^\Sigma \vdash \Sigma_r * [\Delta]$;
16 $\quad\quad\quad\quad \mathsf{R}(\gamma', \mathtt{work}_\mathbf{e}) := \Delta$;
17 $\quad\quad\quad$ **end**
18 $\quad\quad$ **end**
19 $\quad\quad$ **forall the** $\gamma \in \mathcal{C}_\mathsf{N}$ **do**
20 $\quad\quad\quad \mathsf{acquired}(\gamma) := r$;
21 $\quad\quad\quad$ **forall the** γ', γ'' *s.t.* $\gamma \preceq \gamma' \preceq \gamma''$ **do**
22 $\quad\quad\quad\quad \mathsf{N}(\gamma', \gamma'') * [\_] \vdash \Sigma'_r * [\Delta]$;
23 $\quad\quad\quad\quad \mathsf{N}(\gamma', \gamma'') := \Delta$;
24 $\quad\quad\quad$ **end**
25 $\quad\quad$ **end**
26 $\quad\quad \mathcal{C} := \max\{\gamma' \in \mathsf{Paths} \mid \exists \gamma \in \mathcal{C}.\, \mathsf{N}(\gamma, \gamma') = \mathsf{emp}\}$;
27 $\quad$ **end**
28 **end**

the scope of this article. However, to demonstrate how one might compute solutions in practice, we present a heuristic algorithm that works well on our examples.

Computing released and acquired maps. Algorithm 4 constructs released, acquired and resource maps satisfying properties 1–3. Each iteration of the algorithm heuristically picks a needed resource from some subset of program paths, and then iteratively searches for matching redundant resources along all preceding paths. The algorithm maintains a set $\mathcal{C}$ of all program paths up to which no more resources are needed. It terminates once no unsatisfied needed resources remain (line 3).

At the start of the main loop (line 4) the algorithm picks a still-needed resource between a program path in $\mathcal{C}$ and some further program path. The picking of the needed resource is governed by a heuristic function *choose*, which picks a pair of program paths (γ, γ') such that $\gamma \in \mathcal{C}$, $\gamma' \in \mathsf{Paths}$, and $\gamma \prec \gamma'$. We do not make any specific assumption about how these program paths are being picked. For our examples, we introduced a simple partial order reflecting the size of heap resources and picked program paths that lead to largest resources with respect to that order being picked first. We discuss the resource selection heuristic further in Section 7.2.

The key step of the algorithm is performed on line 8.

$$\mathcal{C}_\mathsf{R} := \min\left\{\gamma \in \mathsf{Paths} \,\middle|\, \mathsf{R}(\gamma, \mathtt{work}_\mathbf{e})^\Sigma \vdash \Sigma'_r * [\_] \ \wedge \ \exists \gamma' \in \mathcal{C}_\mathsf{N}.\, \gamma' \preceq \gamma\right\}$$

Here $\mathsf{R}(\gamma, \texttt{work}_{\mathbf{e}})$ is the redundant resource from γ to the end of the `work` function, Σ'_r is the candidate resource that we want to acquire, and $\mathcal{C}_\mathsf{N}$ the set of paths along which the resource Σ_r needs to be acquired. The constructed set $\mathcal{C}_\mathsf{R}$ is a set of program paths along which we can satisfy the candidate needed resource. In line 9, the algorithm checks whether $\mathcal{C}_\mathsf{R}$ covers all paths from $\mathcal{C}_\mathsf{N}$.

Resources stored in needed contain various first-order conditions embedded in the pure part of the symbolic heap. Since we can transfer resources between potentially different program paths, we only take the spatial part of the resource into consideration when asking entailment questions; this is denoted by a superscript Σ. Moreover, since the acquired resource is being sent to a different function invocation, we substitute a fresh set of variables (line 6).

If the check $\mathcal{C}_\mathsf{N} \preceq \mathcal{C}_\mathsf{R}$ succeeds, the remainder of the algorithm is devoted to constructing the new resource (line 11), and updating released (lines 12–18), acquired (lines 19–25), and $\mathcal{C}$ (line 26).

LEMMA 5.4. *Maps* resource*,* released *and* acquired *computed by Algorithm 4 satisfy properties 1–3.*

Consider the run of the Algorithm 4 on our running example (Figure 11 and Figure 12). If *choose* picks (ℓ_1, ℓ_2) in the first iteration and $(\ell_2\ell_3, \ell_2(\ell_2)^{\mathtt{t}}_{\mathbf{e}})$ in the second iteration, then the end result of Algorithm 4 is $\mathsf{resource} = \{r_1 \mapsto (\mathtt{x} \mapsto a), r_2 \mapsto (\mathtt{y} \mapsto b)\}$, $\mathsf{released} = \{\ell_2\ell_3 \mapsto (r_1, \emptyset), \ell_2(\ell_2)^{\mathtt{f}}_{\mathbf{e}} \mapsto (r_1, \emptyset), \ell_2(\ell_2)^{\mathtt{t}}_{\mathbf{e}} \mapsto (r_2, \emptyset), \ell_2\ell_4 \mapsto (r_2, \emptyset)\}$ and $\mathsf{acquired} = \{\ell_1 \mapsto r_1, \ell_2\ell_3 \mapsto r_2, \ell_2\ell_4 \mapsto r_2\}$.

Inserting grant and wait barriers. In this phase we transform the sequential program $\mathbb{P}$ into a parallel program $\mathbb{P}_{par}$ by inserting `grant` and `wait` barriers. The inserted barriers realize the resource transfer established by Algorithm 4 with the maps released and acquired.

We generate the parallel function $\texttt{work}'(\bar{\mathtt{i}}_r^{(p)}, \bar{\mathtt{i}}_r, \texttt{env}^{(p)}, \texttt{env})$ in $\mathbb{P}_{par}$ as follows.

(1) To each $r \in \mathsf{ResId}$ we assign a unique channel name $\mathtt{i}_r$. Denote by $\mathtt{i}_r^{(p)}$ the corresponding channel of the previous thread in the sequence.
(2) Let `env` be an associative array that for each channel maps (escaped) local variable names to values. Let $\texttt{env}^{(p)}$ be such map from the previous thread in the sequence. `env` and $\texttt{env}^{(p)}$ are used for *materialization* (see below).
(3) For each $\gamma \in \mathrm{dom}(\mathsf{acquired})$ such that $\mathsf{acquired}(\gamma) = r$ we insert a wait barrier $\texttt{wait}(\mathtt{i}_r^{(p)})$ between program paths $\mathsf{pred}(\gamma)$ and γ.
(4) For each $\gamma \in \mathrm{dom}(\mathsf{released})$ such that $\mathsf{released}(\gamma) = (r, \_)$, between program paths $\mathsf{pred}(\gamma)$ and γ we insert a sequence of assignments of the form $\texttt{env}(\mathtt{i}_r)[\text{“}\mathtt{y}''] := \mathtt{y}$ for every local variable `y`, followed by a grant barrier $\texttt{grant}(\mathtt{i}_r)$.

Each invocation of $\texttt{work}'$ creates a fresh set of local variables that are bound to the scope of the function. If the structure of some shared resource depends on local variables from a previous invocation, we use the `env` map to transfer the values from the previous thread in a sequence to the next thread in a sequence. The preceding thread *materializes* the variables by storing them in the `env` map (see rule (4) above), which is passed as $\texttt{env}^{(p)}$ to the subsequent thread. Whenever the subsequent thread needs to access a variable from a previous invocation, it refers to the variable using the $\texttt{env}^{(p)}$ map. For instance, the variable `*pr` in the `move()` example from Section 3.2 (Figure 8) would be referred to as $\texttt{env}^{(p)}(\mathtt{i1})[\text{“}\mathtt{p}'']$.

The main function $\texttt{main}_{\mathbb{P}_{par}}$ in $\mathbb{P}_{par}$ first creates the set of “dummy” channels; then in the `while` loop repeatedly creates a set of new channels for the current iteration, forks

a new thread with work′ taking the channels from the previous ($\bar{\mathtt{i}}_r^{(p)}$) and the current ($\bar{\mathtt{i}}_r$) iteration, as well as the maps $\mathtt{env}^{(p)}$ and env, and at the end of the loop body assigns the new channels to the previous channels; and, after the while loop completes, waits on the channels in the last set.

We generate the parallel proof $\mathfrak{P}_{par}$ from the sequential proof $\mathfrak{P}$ using the following specifications for newchan, grant and wait from [Dodds et al. 2011].

$$\{\mathsf{emp}\}\ \ \mathtt{i} := \mathtt{newchan}()\ \ \{\mathsf{req}(\mathtt{i}, R) * \mathsf{fut}(\mathtt{i}, R)\}$$
$$\{\mathsf{req}(\mathtt{i}, R) * R\}\quad \mathtt{grant}(\mathtt{i})\quad \{\mathsf{emp}\}$$
$$\{\mathsf{fut}(\mathtt{i}, R)\}\quad \mathtt{wait}(\mathtt{i})\quad \{R\}$$

The predicates req and fut, corresponding to the required (resp. future) resource, track the ownership of the input and output ends of each channel: $\mathsf{req}(\mathtt{i}, R)$ states that by calling grant on i when holding a resource satisfying R, the thread will lose this resource, and $\mathsf{fut}(\mathtt{i}, R)$ states that by calling wait on i, the thread will acquire a resource satisfying R. In the proof $\mathfrak{P}_{par}$, we instantiate each variable R associated with a channel $\mathtt{i}_r$ with the corresponding resource $\mathsf{resource}(r)$.

Figure 13 illustrates the use of req and fut predicates in the parallel proof of the parallelized function f() from Section 2.

To reason about threads, we use the following standard separation logic rules for fork-join disjoint concurrency [Gotsman et al. 2007].

$$\frac{\{P\}\ \ \mathsf{f}(\bar{\mathtt{x}})\ \ \{Q\}}{\{P[\bar{e}/\bar{\mathtt{x}}]\}\ \ \mathtt{t} = \mathtt{fork}(\mathsf{f}, \bar{e})\ \ \{\mathsf{thread}(\mathtt{t}, \mathsf{f}, \bar{e})\}}\ \text{FORK}$$

$$\frac{\{P\}\ \ \mathsf{f}(\bar{\mathtt{x}})\ \ \{Q\}}{\{\mathsf{thread}(\mathtt{t}, \mathsf{f}, \bar{e})\}\ \ \mathtt{join}(\mathtt{t})\ \ \{Q[\bar{e}/\bar{\mathtt{x}}]\}}\ \text{JOIN}$$

Upon forking a new thread, the parent thread obtains the assertion thread that stores information about passed arguments for program variables and gives up ownership of the precondition of the function. Joining requires that the executing thread owns the thread handle which it then exchanges for the function's postcondition.

THEOREM 5.5. *$\mathfrak{P}_{par}$ is a proof of the parallelized program $\mathbb{P}_{par}$, and defines the same specification for $\mathtt{main}_{\mathbb{P}_{par}}$ as $\mathfrak{P}$ does for $\mathtt{main}_{\mathbb{P}}$.*

Loop-splitting. The approach presented so far treats a loop as a single command with a specification derived from its invariant. Acquiring or releasing resources within a loop is subtle, as it changes the sequential loop invariant. In our algorithm we take a pragmatic approach that performs heuristic loop-splitting of the sequential loop before the parallel code generation so that the conditions guarding acquiring and releasing of the resources become explicit.

The example in Section 3.2 uses two channels to transfer the segment of the list traversed after the first and the second while loop, respectively. The resource released via channel i1 in Figure 8 is $\mathsf{lseg}(\mathtt{s}, y) * \mathtt{pr} \mapsto y$. In the following iteration, the needed resource for the whole loop is $\mathsf{lseg}(\mathtt{s}, \mathtt{p}) * \mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n')$. If we try to match released against needed, the entailment $\mathsf{R}(\gamma, \mathtt{work}_{\mathbf{e}})^{\Sigma} \vdash \Sigma_r' * [\,]$ in Algorithm 4 will fail. This is because the value of p upon exiting the loop cannot be a priori determined, meaning we cannot establish whether the released resource will cover the needed resource.

```
1  {fut(wxp, x ↦ a) ∗ fut(wyp, y ↦ b) ∗ req(wx, x ↦ a′) ∗ req(wy, y ↦ b′)}
2  f(i, wxp, wyp, wx, wy) {
3    wait(wxp);
4   {x ↦ a ∗ fut(wyp, y ↦ b) ∗ req(wx, x ↦ a′) ∗ req(wy, y ↦ b′)}
5    int v = *x;
6   {v = a ∧ x ↦ a ∗ fut(wyp, y ↦ b) ∗
     req(wx, x ↦ a′) ∗ req(wy, y ↦ b′)}
7    if (v >= i) {
8     {v = a ∧ v ≥ i ∧ x ↦ a ∗ fut(wyp, y ↦ b) ∗
      req(wx, x ↦ a′) ∗ req(wy, y ↦ b′)}
9      grant(wx);
10    {v = a ∧ v ≥ i ∗ fut(wyp, y ↦ b) ∗ req(wy, y ↦ b′)}
11     wait(wyp);
12    {v = a ∧ v ≥ i ∗ y ↦ b) ∗ req(wy, y ↦ b′)}
13     *y = v;
14    {v = a ∧ v ≥ i ∗ y ↦ a) ∗ req(wy, y ↦ b′)}
15     grant(wy);
16    {v = a ∧ v ≥ i}
17   }
18   else {
19    {v = a ∧ v < i ∧ x ↦ a ∗ fut(wyp, y ↦ b) ∗
      req(wx, x ↦ a′) ∗ req(wy, y ↦ b′)}
20     wait(wyp);
21    {v = a ∧ v < i ∧ x ↦ a ∗ y ↦ b ∗
      req(wx, x ↦ a′) ∗ req(wy, y ↦ b′)}
22     grant(wy);
23    {v = a ∧ v < i ∧ x ↦ a ∗ req(wx, x ↦ a′)}
24     *x = 0;
25    {v = a ∧ v < i ∧ x ↦ 0 ∗ req(wx, x ↦ a′)}
26     grant(wx);
27    {v = a ∧ v < i}
28   }
29 }
30 {emp}
```

Fig. 13. The separation logic proof of the parallelized function `f()` from Section 2.

One way to resolve this would be to acquire the entire list before the first loop, but this would result in a poor parallelization. Instead, we modify the structure of the loop to expose the point at which the second list segment becomes necessary.

(1) We split the spatial portion of the resource needed by the whole loop into a "dead" (already traversed) and a "live" (still to be traversed) part. In our example, $\mathsf{lseg}(\mathtt{s}, \mathtt{p})$ would be the "dead" and $\mathsf{node}(\mathtt{p}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n') * \mathsf{lseg}(n', \mathsf{null})$ the live part. This kind of splitting is specific to some classes of programs, for instance, linked list programs that do not traverse a node twice.
(2) We match the resource against the "dead" part of the loop invariant and infer the condition under which the two resources are the same. In our example, after unfolding the special cases of list length one and two, the entailment between the two resources holds if $\mathtt{x.nxt} = *\mathtt{pr}$. This condition can be inferred by

```
...
pr:=envp(i1)["p"];
while (i<l && x.nxt!=null && x.nxt!=*pr) {
  i++;
  p=x;
  x=x.nxt;
}
if (i<l && x.nxt!=null && x.nxt==*pr) {
  i++;
  p=x;
  x=x.nxt;
  while (i<l && x.nxt!=null) {
    i++;
    p=x;
    x=x.nxt;
  }
  // remainder skipped.
  ...
else {
...
```

Fig. 14. Loop-splitting for the `move()` example.

ALGORITHM 5: Loop-splitting.

```
1 Split(C: Cmd, c: Π_∨) match C with
2   | case while(b){C'}; C'' →
3   |   | while(b ∧ ¬c'){C'}; if(b ∧ c'){C'; while(b){C'}; C''} else{SPLIT(C'', c)}
4   | case C'; C'' → C'; SPLIT(C'', c);
5   | case skip → skip
6 end
```

asking a bi-abduction question $\mathsf{R}(\gamma, \mathtt{work_e})^{\Sigma} \wedge [c] \vdash \Sigma'_r * [\_]$ with the pure fact c to be abduced.

Now we can syntactically split the loop against the inferred condition $c = (\mathtt{x.nxt} = *\mathtt{pr})$ and obtain a transformed version that ensures that after entering the true branch of the `if` statement the condition c holds. The transformation of our example is shown in Figure 14.

Formally, we define splitting of a command (comprising possibly multiple loops) against a condition c as in Algorithm 5. For simpler exposition, we assume that every command ends with `skip`. The condition c' is obtained from c by replacing a reference to every primed variable $\mathtt{y}'$ by a reference to $\mathtt{env}^{(p)}(\mathtt{i}_r^{(p)})[\text{"y"}]$, where $\mathtt{i}_r$ is the channel name associated to the resource. It is not difficult to see that the accompanying proof of C can be split in a proof preserving way against c. This transformation can either be applied to $\mathbb{P}$ between the resource-usage analysis and parallelization, or embedded within Algorithm 4.

5.3. Implementation

We have validated our parallelization algorithm by crafting a prototype implementation on top of the existing separation logic tool, coreStar [Botinčan et al. 2011]. While our implementation is not intended to provide full end-to-end automated translation, it is capable of validating the algorithms on the examples given in the article, and automatically answering the underlying theorem proving queries.

Our parallelization algorithm does not require a shape invariant generator, except possibly to help construct the sequential proof. Soundness is independent of the "cleanliness" of the invariants (the analysis will always give a correct result, in the worst case defaulting to sequential behavior). The invariants generated automatically by coreStar were sufficient to validate our examples. Other efforts [Calcagno et al. 2011] indicate that bi-abduction works well with automatically generated invariants produced by shape analysis, even over very large code bases.

5.4. Computational Complexity

Deciding entailment of symbolic heaps is solvable in polynomial time for the standard fragment built from the points-to and the inductive linked-list predicate [Cook et al. 2011]. Although a complete syntactic proof search may require an exponential number of proofs to be explored in the worst case [Berdine et al. 2005a], our implementation uses heuristic proof rules that yield polynomial complexity (the potential incompleteness never occurred as a problem in practice). The same procedure is used for the frame inference, the prover just searches for a different kind of leaves in the proof search tree.

Abduction, on the other hand, is a fundamentally harder problem than deduction. While general abduction for arbitrary pure theories reaches the second level of polynomial hierarchy (in fact, it is Σ_2^{P}-complete) [Creignou and Zanuttini 2006; Eiter and Gottlob 1995], abduction of resources for the standard fragment of symbolic heaps is NP-complete. In our implementation we use a polynomial heuristic algorithm (similar to [Calcagno et al. 2011][5]) which has roughly the same runtime cost as frame inference in practice.

Overall, our resource-usage analysis (Algorithm 2 and Algorithm 3) asks $\mathcal{O}(|\mathsf{Paths}|^2 \cdot d \cdot s)$ frame inference and abduction queries, where d is the depth of the abstract syntax tree $\mathcal{T}$ representing the program, and s the arity (the maximum number of siblings) of nodes in $\mathcal{T}$. The heuristic algorithm for computing released and acquired resources (Algorithm 4) asks roughly a cubic number (in the cardinality of Paths) of prover queries.

6. SOUNDNESS OF ANALYSIS

A distinctive property of our parallelizing transformation is that it enforces sequential data dependencies in the parallelized program even if the safety proof does not explicitly reason about such dependencies. The result is that our analysis preserves the sequential behavior of the program: any behavior exhibited by the parallelized program is also a behavior that could have occurred in the original sequential program. However, there are important caveats relating to *termination* and *allocation*.

Termination. If the original sequential program does not terminate, our analysis may introduce new behaviors simply by virtue of running segments of the program that would be unreachable under a sequential schedule. To see this, suppose we have a `pfor` such that the first iteration of the loop will never terminate. Sequentially, the second iteration of the loop will never execute. However, our parallelization analysis will execute all iterations of the loop in parallel. This permits witnessing behaviors from the second (and subsequent) iterations. These behaviors were latent in the original program, and become visible only as a result of parallelization.

[5]The related tool Abductor [Calcagno et al. 2011] has been successfully applied to several code bases counting hundreds of thousands of lines of code.

Allocation and disposal. If the program both allocates and disposes memory, the parallelized program may exhibit aliasing that could not occur in the original program. To see this, consider the following sequential program:

```
x=alloc(); y=alloc(); dispose(x).
```

For simplicity, we have avoided restructuring the program to use data parallelism via `pfor`—this example could easily be encoded as such, however. Parallelization might give us the following program:

```
(x=alloc(); grant(wx); y=alloc()) || (wait(wx); dispose(x)).
```

This parallelized version of the program is race-free and obeys the required sequential ordering on data. Depending upon the implementation of the underlying memory allocator, however, `x` and `y` may be aliased if the `dispose` operation was interleaved between the two allocations. Such aliasing could not happen in the original nonparallelized version.

Either kind of new behavior might result in further new behaviors, for example, we might have an if-statement conditional on `x==y` in the second example above. These caveats are common to our analysis and others based on separation logic, for example, see the similar discussion in Cook et al. [2010].

Behavior preservation. In Section 10 we prove our behavior preservation result (Theorem 10.12). The proof establishes that each trace of the parallelized program corresponds to a trace of the sequential program so that a simulation invariant between the states of the parallel program and the states of the sequential program is preserved.

As will be seen in the proof, one attractive property is that it in fact does not place explicit requirements on the positioning of barriers—it is sufficient that we can provide a separation logic proof of the parallelized version of the program. The caveat on memory disposition manifests in the way we establish our simulation invariant, which necessarily assumes newly allocated data is always "fresh" and thus does not alias with any accessible heap-allocated structure. The caveat on termination is necessary to ensure we can suitably reorder threads in the parallelized program to yield a "sequentialized" trace.

In addition to inserting barriers, our analysis mutates the program by materializing thread-local variables and splitting loops. Both of these can be performed as mutations on the initial sequential program, and neither affect visible behavior. Loop-splitting is straightforwardly semantics-preserving, while materialized variables are only used to control barrier calls.

Theorem 10.12 guarantees that parallelization does not introduce deadlocks; otherwise the simulation relation would not exist. The ordering on barriers ensures that termination in the sequential program is preserved in the resulting parallelized program.

7. ADJUSTING THE ANALYSIS

Our whole approach, assuming a fully automated prover, can be seen as being parametric in (1) abstract domain for resources, (2) resource selection procedure, and (3) the level of path-sensitivity. In this section, we discuss the way the properties of the analysis change under various choices of these parameters.

7.1. Resource Domain

Our analysis is generic in the choice of abstract domain; any separation logic predicate can potentially be used in place of lseg, for example. The choice of resource predicates affects the success of the analysis particularly strongly, as resources can only

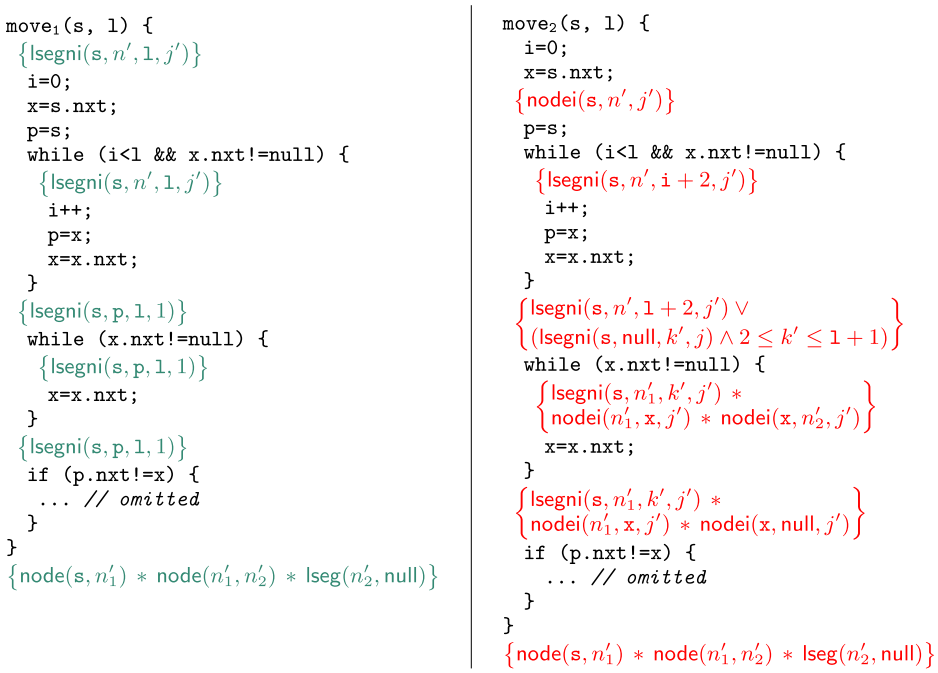


Fig. 15. Redundant and needed resources for `move()` calculated using the lsegni resource predicate.

be accessed in parallel if they can be expressed disjointly. Stronger domains allow the analysis to split resources more finely, and so (in some cases) making the parallelization better. However, altering the choice of resources may also force barriers earlier, making the parallelization worse.

In Section 3.2 we parallelized the `move()` example using the resource predicates lseg, representing list segments, $x \mapsto a$, representing individual heap cells, and $x \overset{c}{\mapsto} a$, representing fractional ownership of a cell. To see how the choice of abstract domain influences the analysis, in this section, we consider two other resource predicates: list segments with length; and list segments with membership.

List segment with length. The predicate $\mathsf{lsegni}(h,t,n,i)$ extends lseg with a parameter $n \in \mathbb{Z}$ recording the *length* of the list segment, and with a parameter $i \in (0,1]$ recording the *permission* on the list segment. If $i = 1$ the thread has read-write access; otherwise it has read access only (this follows the behavior of permission on individual heap cells). We define nodei and lsegni as follows.

$$\mathsf{nodei}(x,y,i) \triangleq x.\mathrm{val} \overset{i}{\mapsto} v' * x.\mathrm{nxt} \overset{i}{\mapsto} y$$

$$\mathsf{lsegni}(x,t,n,i) \triangleq (x = t \wedge n = 0 \wedge \mathsf{emp}) \vee (\mathsf{nodei}(x,y',i) * \mathsf{lsegni}(y',t,n-1,i))$$

We use as input proof the original sequential proof shown in Figure 7. The equivalence $\mathsf{lseg}(h,t) \iff \mathsf{lsegni}(h,t,n',1)$ allows the analysis to exploit the lsegni predicate even though the sequential proof is written using the coarser lseg predicate.

We will now use the lsegni and node predicates when parallelizing our running example `move()` function. Once again, we parallelize a pair of calls to `move()`.

```
move(s,a); move(s,b);
```

```
move1(s, l) {
  i=0;
  x=s.nxt;
  p=s;
  *lr=l;
  grant(i1);
  while (i<l && x.nxt!=null) {
    i++;
    p=x;
    x=x.nxt;
  }
  while (x.nxt!=null)
    x=x.nxt;
  if (p.nxt!=x) {
    ... // omitted
  }
  grant(i2);
}
```

```
move2(s, l) {
  i=0;
  wait(i1);
  if (2 >= *lr) wait(i2);
  x=s.nxt;
  p=s;
  while (i<l && x.nxt!=null) {
    if (i+2 >= *lr) wait(i2);
    i++;
    p=x;
    x=x.nxt;
  }
  if (l+2 < *lr) wait(i2);
  while (x.nxt!=null) {
    x=x.nxt;
  }
  if (p.nxt!=x) {
    ... // omitted
  }
}
```

Fig. 16. Parallelization of `move()` using lsegni.

The needed and redundant resources for the first and second calls to move calculated using lsegni are shown in Figure 15.

In first call to `move()`, the resource lsegni(s, n', l, j') is redundant immediately. This is because `move()` only needs read-write access to the list at the `l`-th node. It can pass read-access to the second call to `move()`. Nodes earlier than `l` in the list can immediately be accessed by the subsequent call.

Conversely, the while-loops in `move()` only need read-only access to the list. This is reflected in the fact that the needed resources all use lsegni. It is only in the final if-statement that `move()` requires write-access to the list.

Figure 16 shows a parallelization of the example using lsegni. There are several changes over the parallelization with lseg that we show in Figure 8.

— The value `lr` shared between $\texttt{move}_1\texttt{()}$ and $\texttt{move}_2\texttt{()}$ has been changed from the address of a node, to the position of the node to be moved.
— In $\texttt{move}_2\texttt{()}$, the calls to `wait(i2)` are conditional on the position value stored in `lr`, rather than on the address of the `.nxt` node.
— In $\texttt{move}_1$ the call to `grant(i1)` has been pushed up past the first while-loop.
— The third call to `wait(i2)` has been moved up out of the while loop.

The first two changes make no difference to the degree of parallelization. The third, moving the call to `grant()` upwards, makes the program more parallel: the subsequent call to $\texttt{move}_2\texttt{()}$ needs to wait less time before it can proceed. The change in domain enables this optimization because it allows the analysis to represent a list segment of a particular fixed length, and to work out that it can be safely transferred to the second call to `move()`.

The fourth change, moving the `wait()` upwards, makes the program less parallel. This is a consequence of sharing the position of the node, rather than its address. In the original parallelization, the loop could conditionally wait on the address of the node. However, the second loop does not store it's position in the list, meaning that it cannot check when it needs to call `wait()`. To be conservative, it must wait for the resource *before* entering the loop.

```
build(){
  s=malloc();
  x=s;
  do {
    x.val=0;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  do {
    x.val=1;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  do {
    x.val=2;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  x.val=3;
  return s;
}
```

```
check(s){
  x=s;
  while(x != null) {
    if (x.val==1)
      return true;
  }
  return false;
}
```

Fig. 17. Functions `build()`, which constructs a linked list containing values $0^+1^+2^+3$, and `check()`, which checks whether a 1-valued node is present.

Thus, there is a subtle interplay between the choice of domain, the particular resources chosen by the analysis, and the parallelization that can be achieved. (Of course, as the lsegni domain includes the original lseg domain, the analysis could also construct the original parallelization). However, simply making the domain richer does not automatically make the parallelization more successful.

List segment with membership. The list predicates we have considered so far have remembered the structure of the list, but not the values held in it. However, in some cases we can get a better parallelization if the predicate records the values stored in the list. To do this, we use the predicates nodev and lsegv. Respectively, these represent a list node holding a particular value, and a list segment with an associated set of values.

$$\begin{aligned} \mathsf{nodev}(x,y,v) &\triangleq x.\mathrm{val} \mapsto v * x.\mathrm{nxt} \mapsto y \\ \mathsf{lsegv}(x,t,S) &\triangleq (x = t \wedge S = \emptyset \wedge \mathsf{emp}) \vee \\ &\quad (\mathsf{nodev}(x,y',v') * (\mathsf{lsegv}(y',t,S \setminus \{v'\}) \vee \mathsf{lsegv}(y',t,S))). \end{aligned}$$

Note that in $\mathsf{lsegv}(x,y,S)$ the set of values S is a subset of all the values in the list: if a value is in S, then it must be in the list, but not vice versa.

We apply nodev and lsegv to the functions `build()` and `check()`, defined in Figure 17. The `build()` function constructs a linked list a sequence of values matching the regular expression $0^+1^+2^+3$. The `check()` function reads through the list and returns true if a 1-value node is present, and zero otherwise. This example is inspired by similar build-and-check examples defined in the SV-Comp benchmark suite [SV-Comp 2013].

We can parallelize the following program.

```
r = build(); check(r);
```

The possible parallelizations depend on the needed and redundant resources calculated by the analysis. These in turn depend on the choice of resource predicates. The

```
build(){
  s=malloc();
  x=s;
  do {
    x.val=0;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
 1. {node(s, n′) ∗ lseg(n′, x)}
 2. {lsegv(s, x, {0})}
  do {
    x.val=1;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
 1. {node(s, n′) ∗ lseg(n′, x)}
 2. {lsegv(s, x, {0, 1})}
  do {
    x.val=2;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  x.val=3;
 1. {node(s, n′) ∗ lseg(n′, null)}
 2. {lsegv(s, x, {0, 1, 2, 3})}
  return s;
}
```

```
check(s){
  x=s;
 1. {node(s, n′)}
 2. {node(s, n′)}
  while(x != null) {
    if (x.val==1)
      return true;
  }
 1. {node(s, n′) ∗ lseg(n′, null)}
 2. {(node(s, n′) ∗ lseg(n′, null)) ∨ lsegv(s, n′, {1})}
  return false;
}
```

Fig. 18. Redundant and needed resources for `build()` and `check()`. Resources labeled (1) are calculated with the default lseg predicate, while resources labeled (2) are calculated with the lsegv predicate.

needed and redundant resources calculated without and with lsegv are shown interleaved in the code in Figure 18.

Suppose we only use the lseg and node predicates: then the analysis can only establish that the resources redundant in `build()` are list segments, without expressing any knowledge of the values stored. With lseg, the analysis calculates that `check()` requires access to the whole list. If we use lsegv, then the analysis computes that redundant resources in `build()` are list segments containing particular values. It also calculates that `check()` either needs access to a list segment containing `1`, or a complete list segment—either will suffice.

The difference between parallelizations can be seen in Figure 19. To make the exposition clearer, we only inject barriers into loop-free code, rather than injecting barriers into the loop. On the left, we have the parallelization possible using lseg. The `grant`-barrier in `build()` waits until after the final loop has terminated. On the right, we have the parallelization with lsegv. The analysis is able to work out that `check()` only needs access to a list segment containing a `1`-valued node. This needed resource be satisfied after the second loop. The `grant`-barrier can thus been moved upwards above the third loop, resulting in a better parallelization.

7.2. Resource Selection Procedure

Algorithm 4 is parameterized by a function *choose* governing resource selection. In a way, *choose* can be seen as a proxy for external knowledge (e.g., hints provided by a programmer) about likely points for parallelization. The way how *choose* picks program paths determining the resource selection is not important for the correctness of the algorithm, but different strategies can lead to better or worse parallelizations.

```
build(){
  s=malloc();
  x=s;
  do {
    x.val=0;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  do {
    x.val=1;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  do {
    x.val=2;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  x.val=3;
  grant(i);
  return s;
}
```

```
check(s){
  x=s;
  wait(i);
  while(x != null) {
    if (x.val==1)
      return true;
  }
  return false;
}
```

```
build(){
  s=malloc();
  x=s;
  do {
    x.val=0;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  do {
    x.val=1;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  grant(i);
  do {
    x.val=2;
    x.nxt=malloc();
    x=x.nxt;
  } while(nondet);
  x.val=3;
  return s;
}
```

```
check(s){
  x=s;
  wait(i);
  while(x != null) {
    if (x.val==1)
      return true;
  }
  return false;
}
```

Fig. 19. Left: parallelization of `build()` and `check()` using the lseg predicate. Right: parallelization using lsegv.

In our examples, we used a very simple, automatic greedy heuristic for the *choose* function. We introduced a simple syntactic partial order $\lhd$ reflecting the coarse-grained size of heap resources, under which:

$$\begin{aligned} \mathsf{emp} &\lhd e \mapsto f \\ \mathsf{emp} &\lhd \mathsf{lseg}(e,f) \\ e \mapsto f &\lhd \mathsf{lseg}(e',f') \wedge e' \neq f', \end{aligned}$$

and $\Sigma \lhd \Sigma'$ if for every conjunct S in Σ there exists a conjunct S' in Σ' such that $S \lhd S'$. We write $\Sigma \bowtie \Sigma'$ if $\Sigma \lhd \Sigma'$ and $\Sigma' \lhd \Sigma$. Then, given a set of program path pairs $\{(\gamma,\gamma') \mid \gamma \in \mathcal{C} \wedge \gamma' \in \mathsf{Paths} \wedge \gamma \prec \gamma'\}$, we defined a partial order $\sqsubset$ by letting $(\gamma,\gamma') \sqsubset (\delta,\delta')$ if $\mathsf{N}(\delta,\delta') \lhd \mathsf{N}(\gamma,\gamma')$ or $\mathsf{N}(\gamma,\gamma') \bowtie \mathsf{N}(\delta,\delta')$, $\gamma = \delta$ and $\gamma' \prec \delta'$. The *choose* function traversed topologically sorted program path pairs and picked each pair in turn.

This *choose* function favored picking "large" resource that are encountered along program paths first, which worked well for our examples. However, in some cases we might produce better parallelizations by picking "smaller" resources first. Such preference of resources can be tuned by redefining the partial order $\lhd$.

7.3. Path-Sensitivity

While the resource-usage analysis (Section 5.1) is fully path-sensitive (modulo loops), the parallelizing transformation (Section 5.2) accounts for path-sensitivity only syntactically. During the execution of the parallelized program precisely one call to `wait()` and `grant()` should occur for each channel. However, barriers may be injected into conditional branches, creating the possibility of missed barriers or unwanted multiple calls to barriers. To ensure that barriers are called exactly once, the parallelizing transformation follows the syntactic structure of the program, forcing the calls to `grant()` to occur at program paths that are covering the corresponding calls to `wait()` for the same channel (the condition $\mathcal{C}_\mathsf{N} \preceq \mathcal{C}_\mathsf{R}$ in line 9 of Algorithm 4). This approach is sufficient for all of the examples that we consider.

We could strengthen this approach by making the prover record explicit path conditions in the assertions of the sequential proof and using these conditions semantically. To check that all paths are covered by the points chosen for barrier injection, Algorithm 4 could then disjoin the path conditions and check for tautology. More precisely, if we let $\mathsf{pc}\colon \mathsf{Paths} \to \Pi_\vee$ represent the path condition associated with each program path, then we would replace the condition in line 9 of Algorithm 4 with $\bigvee_{\gamma \in \mathcal{C}_\mathsf{R}} \mathsf{pc}(\gamma) \Leftrightarrow \mathsf{true}$. To check that two paths cannot be satisfied at the same time, the algorithm could check that the associated path conditions contradict.

Enriching the parallelizing transformation in this way would allow a possibly better placement of barriers down conditional branches. For example, the following barrier placement would be forbidden by the syntactic approach, but allowed by the semantic path-sensitive version.

```
if (P)          |   if (Q)
  grant(i);     |     wait(i);
if (¬P)         |   if (¬Q)
  grant(i);     |     wait(i);
```

8. LIMITATIONS

Loop handling. Our parallelizing transformation treats loops as opaque, with a loop-splitting transformation used to allow mid-loop signaling. Since not all loops have a structure amenable for such a transformation, it will not always be possible to discover parallelization opportunities within loop-based computations. However, one can envision analogous transformations that would split according to different phase-switching patterns in the loop body.

In the main parallel-`for` loop, we currently assume each iteration communicates with only its immediate predecessor and successor. A technique called *renunciation*, discussed in Dodds et al. [2011], allows a thread to release a resource without having to first acquire it, allowing resource transfers to skip iterations. It would be relatively straightforward to fold such a technique into our analysis.

Procedures. The analysis we given in the body of the article treats procedures in two ways: either as inlined code, or opaquely, in terms of their sequential specifications. The latter lets us deal with recursively defined procedures, although it means that such procedures cannot have barriers injected into them. This is a similar simplification to treating loops as opaque. As with loops, we can develop a more sophisticated procedure-unrolling transformation which would exploit parallelization opportunities across function calls. We present such an inter-procedural version of the analysis in the Online Appendix.

Achieving scalability. A naïve implementation of our proposed algorithm would not scale due to rapid combinatorial explosion of program paths. However, the algorithm can be tuned to avoid exploring paths that are unlikely to expose parallelism or, conversely, to explore paths that involve intense computations worth running concurrently. We note that reducing the depth of the abstract syntax tree unwinding only limits the potential for parallelization but does not lead to unsoundness.

Tool support. An automated separation logic prover for solving frame inference, abduction and bi-abduction queries is essential for our approach, and the quality of parallelization depends strongly on the success of the prover. However, our analysis degrades gracefully with imprecise abduction giving less precise parallelization. In the worst case, we obtain sequentialization of the parallel threads, reflecting our

analysis being unable to prove that any other parallelization was safe with respect to our ordering assumptions.

Nonlist domains. As discussed in Section 7, our analysis is generic in the choice of the abstract domain; any separation logic predicate could be used in place of lseg, for example. However, the success of *automated* parallelization is highly dependent on the power of the entailment prover in the chosen domain. The lseg domain is one of the best developed in separation logic, and consequently automated parallelization is feasible using tools such as `coreStar`. Other domains (such as trees) are far less developed so additional automated reasoning support and special-purpose heuristics for abduction may be needed.

Alternative parallelization primitives. Our use of `pfor` as our primitive parallelization construct in Section 3.1 is a design choice, rather than an essential feature of our algorithm. We could use a more irregular concurrency annotation, for example safe futures [Navabi et al. 2008]. In this case, our resource-usage analysis would be mostly unchanged, but our parallelized program would construct a set of syntactically distinct threads, rather than a pipeline of identical threads.

We emphasize that our parallelization algorithm is targetting programs where ordering is important. As the analysis is syntactically driven by the structure of the loop and the signaling order, it may impose order between iterations that are not strictly enforced by the underlying data structure. For instance, if we would use unordered parallel-`for` (e.g., as in the Galois system [Pingali et al. 2011]) our current analysis would generate a deterministic parallel program that respects some specific sequential order of the loop iterator (contrary to Galois that would, with its speculative parallel execution, allow nondeterminism). Removing ordering between iterations could be achieved by replacing ordered `grant-wait` pairs with conventional locks, but this would introduce an extra obligation to show that locks were always acquired as a set by a single thread at once.

9. SEMANTICS

We proceed to define an operational semantics of the multithreaded core language to which the programs generated by our parallelizing transformation are translated; we then prove a behavior-preservation result for our analysis. The syntax of this language is similar to the sequential core language described in Section 4.1, but is additionally equipped with threads and operations on channels (`newch`, `grant`, and `wait`).

Our semantics makes several simplifying assumptions: (1) We forbid disposal, and so can assume that allocation always gives fresh locations. This restriction is introduced because full memory allocation (with memory disposition and reuse) may allow parallelization to introduce new behaviors (see Section 6); (2) to simplify the presentation, our semantics does not include function definitions or calls. Data-races are interpreted as faults in this semantics.

The operational semantics has two levels, a labeled thread-local semantics ('$\rightsquigarrow$'), and a global semantics ('$\Longmapsto$') defined in terms of the local semantics. Every transition in the global semantics corresponds to a transition for some thread in the local semantics. Our semantics is *annotated* in the sense that it keeps track of the resources that are associated with threads and channels. These distinctions would not be present in the operational semantics of the real machine.

Thread-local semantics. The semantics assumes the following basic sets: Prog, programs; Tid, thread identifiers, ordered by $<$; Cid, channel identifiers; Var, variable names; Loc, heap locations; Val, primitive values (which include locations). We assume that the sets Tid, Cid and Loc are infinite, and that $\mathsf{Tid} \uplus \mathsf{Cid} \uplus \mathsf{Loc} \subseteq \mathsf{Val}$.

$$\frac{(\sigma,\sigma') \in c}{(c,\sigma,\omega,\gamma) \rightsquigarrow (\mathbf{skip},\sigma',\omega,\gamma)} \qquad \frac{\neg\exists\sigma'.\,(\sigma,\sigma') \in c}{(c,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{\sigma,\omega \not\models P * \mathsf{true}}{(\mathbf{fork}_{[P]}C,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}} \qquad \frac{\sigma',\omega' \models P \quad \sigma = \sigma' \oplus \sigma'' \quad \omega = \omega' \uplus \omega'' \quad t \text{ fresh}}{(\mathbf{fork}_{[P]}C,\sigma,\omega,\gamma) \overset{\mathbf{fork}\,(t,C,\sigma',\omega')}{\rightsquigarrow} (\mathbf{skip},\sigma'',\omega'',\gamma \uplus \{t\})}$$

$$\frac{s \text{ fresh}}{(x := \mathbf{newch},\sigma,\omega,\gamma) \overset{\mathbf{newch}\,(s)}{\rightsquigarrow} (\mathbf{skip},(\sigma_s[x \mapsto s],\sigma_h),(\omega_{\mathrm{w}} \uplus \{s\},\omega_{\mathrm{g}} \uplus \{s\}),\gamma)}$$

$$\frac{\sigma,(\emptyset,\emptyset) \not\models P * \mathsf{true}}{(\mathbf{grant}_{[P]}E,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}} \qquad \frac{[\![E]\!]_{\sigma_{\mathrm{s}}} = s \quad s \in \omega_{\mathrm{g}} \quad \sigma = \sigma' \oplus \sigma'' \quad \sigma',(\emptyset,\emptyset) \models P}{(\mathbf{grant}_{[P]}E,\sigma,\omega,\gamma) \overset{\mathbf{grant}\,(s,\sigma')}{\rightsquigarrow} (\mathbf{skip},\sigma'',(\omega_{\mathrm{w}},\omega_{\mathrm{g}} \setminus \{s\}),\gamma)}$$

$$\frac{[\![E]\!]_{\sigma_{\mathrm{s}}} = s \quad s \in \omega_{\mathrm{w}}}{(\mathbf{wait}\,E,\sigma,\omega,\gamma) \overset{\mathbf{wait}\,(s,\sigma')}{\rightsquigarrow} (\mathbf{skip},\sigma \oplus \sigma',(\omega_{\mathrm{w}} \setminus \{s\},\omega_{\mathrm{g}}),\gamma)}$$

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow (C',\sigma',\omega',\gamma')}{(C;C'',\sigma,\omega,\gamma) \rightsquigarrow (C';C'',\sigma',\omega',\gamma')} \qquad \frac{}{(\mathbf{skip};C,\sigma,\omega,\gamma) \rightsquigarrow (C,\sigma,\omega,\gamma)}$$

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}{(C;C',\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}$$

$$\frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{tt}}{(\mathbf{if}\,(B)\,C_1\,\mathbf{else}\,C_2,\sigma,\omega,\gamma) \rightsquigarrow (C_1,\sigma,\omega,\gamma)} \qquad \frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{ff}}{(\mathbf{if}\,(B)\,C_1\,\mathbf{else}\,C_2,\sigma,\omega,\gamma) \rightsquigarrow (C_2,\sigma,\omega,\gamma)}$$

$$\frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{tt}}{(\mathbf{while}\,(B)\,C,\sigma,\omega,\gamma) \rightsquigarrow (C;\mathbf{while}\,(B)\,C,\sigma,\omega,\gamma)} \qquad \frac{[\![B]\!]_{\sigma_{\mathrm{s}}} = \mathsf{ff}}{(\mathbf{while}\,(B)\,C,\sigma,\omega,\gamma) \rightsquigarrow (\mathbf{skip},\sigma,\omega,\gamma)}$$

$$\frac{l \notin \mathrm{dom}(\sigma_h)}{(x := \mathbf{alloc},\sigma,\omega,\gamma) \overset{\mathbf{alloc}(l)}{\rightsquigarrow} (\mathbf{skip},(\sigma_s[x \mapsto l],\sigma_h \uplus [l \mapsto 0]),\omega,\gamma)}$$

Fig. 20. Annotated thread-local operational semantics.

The thread-local semantics, whose rules are given in Figure 20, is defined by a labeled transition relation $\rightsquigarrow \in \mathcal{P}(\mathsf{TState} \times \mathsf{TState} \times \mathsf{Label})$, with TState defined as follows.

$$\begin{aligned}
\sigma_s \in\ & \mathsf{Stack} : \mathsf{Var} \rightharpoonup \mathsf{Val} \\
\sigma_h \in\ & \mathsf{Heap} : \mathsf{Loc} \rightharpoonup \mathsf{Val} \\
\sigma \in\ & \mathsf{LState} : \mathsf{Stack} \times \mathsf{Heap} \\
(\omega_{\mathrm{w}},\omega_{\mathrm{g}}) \in\ & \mathsf{CState} : \mathcal{P}(\mathsf{Cid}) \times \mathcal{P}(\mathsf{Cid}) \\
& \mathsf{TState} : \mathsf{Prog} \times \mathsf{LState} \times \mathsf{CState} \times \mathcal{P}(\mathsf{Tid})
\end{aligned}$$

The semantics associates each thread with a resource which can be safely manipulated by that thread. Globally visible events, meaning forking, creating channels, granting and waiting, and heap allocation are modeled by labels on the thread-local transition relation. These labels are used to ensure that global events are propagated to all threads. The set of labels, Label, is defined as follows.

$$\begin{aligned}
\mathsf{Label} \triangleq\ & (\{\mathbf{fork}\} \to \mathsf{Tid} \times \mathsf{Prog} \times \mathsf{LState} \times \mathsf{CState}) \\
& \uplus (\{\mathbf{newch}\} \to \mathsf{Cid}) \\
& \uplus (\{\mathbf{wait},\mathbf{grant}\} \to \mathsf{Cid} \times \mathsf{Heap}) \\
& \uplus (\{\mathbf{alloc}\} \to \mathsf{Loc})
\end{aligned}$$

The rules use a join operator on heaps $\oplus\colon \mathsf{Heap} \times \mathsf{Heap}$ by union of functions, defined only if the two heaps' domains are disjoint. The join operator $\oplus\colon \mathsf{LState} \times \mathsf{LState} \rightharpoonup \mathsf{LState}$ is defined as identity on stacks and join on heaps:

$$\sigma \oplus \sigma' \triangleq (\sigma_s, \sigma_h \oplus \sigma'_h) \qquad \text{if } \sigma_s = \sigma'_s.$$

A *local state* in LState consists of a *stack* mapping variables to values and a *heap* mapping locations to values. Note that we do not support address arithmetic. A *channel state* in CState consists of a pair of sets, respectively recording the channels the thread can **wait** for and **grant** on. A *thread state* in TState consists of a program (i.e., command), a local state, a channel state, and a set of thread identifiers of child threads.

The semantics assumes a set of primitive commands $\mathsf{Prim}\colon \mathcal{P}(\mathsf{LState} \times \mathsf{LState})$. We assume the following for any command $c \in \mathsf{Prim}$.

— For any states σ_1 and σ_2 such that $(\sigma_1 \oplus \sigma_2, \sigma') \in c$, if there exists a transition $(\sigma_1, \sigma'_1) \in c$ then $\sigma'_1 \oplus \sigma_2 = \sigma'$.
— For any transition $(\sigma, \sigma') \in c$ and state σ_2, if $\sigma \oplus \sigma_2$ is well-defined, then there exists a transition $(\sigma \oplus \sigma_2, \sigma'') \in c$ and $\sigma'' = \sigma' \oplus \sigma_2$.
— For any states σ_1 and σ_2, if there exists no σ' such that such that $(\sigma_1 \oplus \sigma_2, \sigma') \notin c$, then $(\sigma_1, \sigma') \notin c$.

Besides capturing a semantic relation between local states, primitive commands also have a syntactic representation (e.g., the name of the command itself). We overload these notions in the rules to allow us flexibility in characterizing effects without having to choose a fixed command set *a priori*. It is straightforward to define an interpretation function that maps the nominal representation of a command with its relational definition.

The effect of evaluating primitive command c in state $\langle c, \sigma, \omega, \gamma\rangle$ is a new state $\langle \mathbf{skip}, \sigma', \omega, \gamma\rangle$ if $(\sigma, \sigma') \in c$. If command c can effect no transition from local state σ, the thread aborts.

A thread may fork a child. We associate a syntactic assertion P with a **fork** command that captures the resources required by the child thread. (The existence of a syntactic separation logic proof means we can always annotate a *fork* command with such assertions.) If the assertion is not satisfiable within the current local state (i.e., $\sigma, \omega \not\models P * \mathsf{true}$), the thread aborts. Otherwise, we can partition the state, associating the resources (memory and channels) needed by the forked thread (denoted as σ' and ω' in the rule) from those required by the parent. The effect annotation $\mathbf{fork}\ (t, C, \sigma', \omega')$ records this action. We assume that the fresh thread identifier t created on each **fork** invocation is strictly greater (with respect to $<$) than previous ones.

Creating a new channel with **newch** augments the set of channel identifiers for both **wait** and **grant** actions.

In order for a thread to **grant** resources to another thread, the assertion that defines the structure of these resources must hold in the current local state; note that the only resources that can be propagated along channels are locations and values in the thread's local state—channel identifiers can only be distributed at fork-time. If the assertion indeed holds, then the resources it describes can be transferred on the channel. This semantics ensures all concurrently executing threads operate over disjoint portions of shared memory. The semantics of **wait** transfers a set of resources to the executing thread whose composition is determined by the assertion on the corresponding **grant** that communicates on channel s. The rules for sequencing and loops are standard.

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow (C',\sigma',\omega',\gamma')}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta,\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{fork}\,(t_2,C_2,\sigma_2,\omega_2)}{\rightsquigarrow} (C',\sigma',\omega',\gamma')}{([t_1 \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow ([t_1 \mapsto C',\sigma',\omega',\gamma'] \uplus [t_2 \mapsto C_2,\sigma_2,\omega_2,\emptyset] \uplus \delta,\eta,\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{newch}\,(s)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad s \notin \mathrm{dom}(\eta)}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta \uplus [s \mapsto \triangle],\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{grant}\,(s,\sigma)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad \eta(s) = \triangle}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta[s \mapsto \sigma],\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{wait}\,(s,\sigma)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad \eta(s) = \sigma}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta[s \mapsto \nabla],\mathcal{L})}$$

$$\frac{(C,\sigma,\omega,\gamma) \overset{\mathbf{alloc}(l)}{\rightsquigarrow} (C',\sigma',\omega',\gamma') \quad l \in \mathcal{L}}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow ([t \mapsto C',\sigma',\omega',\gamma'] \uplus \delta,\eta,\mathcal{L} \setminus \{l\})}$$

$$\frac{(C,\sigma,\omega,\gamma) \rightsquigarrow \mathbf{abort}}{([t \mapsto C,\sigma,\omega,\gamma] \uplus \delta,\eta,\mathcal{L}) \Longrightarrow \mathbf{abort}}$$

Fig. 21. Annotated global operational semantics.

Memory allocation annotates the transition with an allocation label $\mathbf{alloc}(l)$ for fresh location l.

Global semantics. The global semantics is defined as a transition relation $\Longrightarrow \in \mathcal{P}(\mathsf{GState} \times \mathsf{GState})$, with GState defined as follows.

$$\begin{aligned} \delta &\in \ \mathsf{TMap} : \mathsf{Tid} \rightharpoonup \mathsf{TState} \\ \eta &\in \ \mathsf{CMap} : \mathsf{Cid} \rightharpoonup \mathsf{Heap} \uplus \{\triangle, \nabla\} \\ \kappa &\in \ \mathsf{GState} : \mathsf{TMap} \times \mathsf{CMap} \times \mathcal{P}(\mathsf{Loc}) \end{aligned}$$

The rules of the global semantics are given in Figure 21. Every transition in the global semantics corresponds to a transition for some thread t in the local semantics. The global semantics models the pool of active threads with a thread-map in TMap, and models the resources held by channels with a channel-map in CMap. The set of locations in $\mathcal{P}(\mathsf{Loc})$ represents unallocated locations in the heap. Because we never return locations to this set, all allocated locations are fresh.

Given a channel map $\eta \in \mathsf{CMap}$, a given channel $s \in \mathrm{dom}(\eta)$ can be either allocated but unused, denoted by $\eta(s) = \triangle$, in use, denoted by $\eta(s) \in \mathsf{Heap}$, or finalized, denoted by $\eta(s) = \nabla$. Once finalized, a channel identifier cannot be reused (fortunately we have an infinite supply of fresh channel identifiers). The semantics enforces consistency between calls to **wait**, **grant**, and **newchan**. A thread can only take a local transition acquiring or releasing a resource if it is consistent with the global channel map.

Thread-local steps are mirrored in the global semantics by updating the state of the thread in the thread map. A fork operation augments the thread map by associating the thread identifier with a new thread-local configuration whose components are specified on the local transition. Creating a new channel is permissible only if the

channel identifier does not already exist. A **grant** action is allowed if the channel is unused; after the action the channel is associated with the heap resources provided by the **grant**. A **wait** action is allowed if the resources demanded by the **wait** are provided by the channel (via a **grant**); if so, the channel becomes finalized. Allocation removes the location from the set of available locations.

Definition 9.1. For a global state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ and thread $t \in \mathrm{dom}(\delta)$ such that $\delta(t) = \langle C, \sigma, (\omega_{\mathrm{w}}, \omega_{\mathrm{g}}), \gamma \rangle$, we define $\mathrm{waits}(\kappa, t) \triangleq \omega_{\mathrm{w}}$ and $\mathrm{grants}(\kappa, t) \triangleq \omega_{\mathrm{g}}$.

Definition 9.2 (*well-formedness*). A global state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ is *well-formed* if:

(1) The set of all heap locations either allocated and referred to in δ and η is disjoint from $\mathcal{L}$, the set of unallocated heap locations.
(2) The thread-local states in δ and states stored in the channel map η can be joined using $\oplus$ to give a well-defined state. Note that, as $\oplus$ is associative and commutative, this state is unique, and also that any pair of states from δ and η can be joined to give a well-defined result.
(3) For any channel $c \in \mathsf{Cid}$, there exists at most one thread t_1 such that $c \in \mathrm{grants}(\kappa, t_1)$ and at most one thread t_2 such that $c \in \mathrm{waits}(\kappa, t_2)$. If $c \in \mathrm{grants}(\kappa, t_1)$, then $\eta(c) = \triangle$. If $c \in \mathrm{waits}(\kappa, t_2)$, then $\eta(c) \in \mathsf{Heap} \uplus \{\triangle\}$.

Well-formedness expresses fundamental linearity assumptions on states and channels; essentially, it ensures elements of the state can be owned by only one thread at once.

LEMMA 9.3. *The global transition relation* $\Longmapsto$ *preserves well-formedness.*

Assumption 1. We assume all global states are well-formed, unless explicitly stated otherwise.

Definition 9.4 (*trace*). A trace is a (finite or infinite) sequence of global states $\mathcal{K} = \kappa_0 \kappa_1 \ldots$ such that for every i, $\kappa_i \Longmapsto \kappa_{i+1}$. A finite (infinite) trace is called terminating (nonterminating).

Definition 9.5 (*child thread, thread order*). Each local state $\langle C, \sigma, \omega, \gamma \rangle$ includes the set γ of forked child thread identifiers. For convenience, we define $\mathrm{child}(\kappa, t)$ as the set of children for thread t in global state κ. The children of a thread are ordered with a relation $<$ that follows the order in which they were created. That is, suppose we have a trace $\mathcal{K}$ with initial state $\langle \delta, \eta, \mathcal{L} \rangle$ such that $c_1, c_2 \notin \mathrm{dom}(\delta)$. Given an arbitrary state $\kappa_i \in \mathcal{K}$, if $\{c_1, c_2\} \in \mathrm{child}(\kappa, t)$ and $c_1 < c_2$, then c_1 was created earlier in the trace than c_2.

We denote by $\Longmapsto^*$ the reflexive transitive closure of the global transition relation. We sometimes write '$\overset{t}{\Longmapsto}$' to denote a global transition resulting from the thread t taking a step, and call it a transition over thread t.

Definition 9.6 (*sequentialized trace*). We define the *sequentialized* transition relation $\Longmapsto_{\mathrm{s}} \subseteq \Longmapsto$ as the transition relation in which a step $\kappa \overset{t}{\Longmapsto} \kappa'$ can only be taken if all threads in $\mathrm{child}(\kappa, t)$ have reduced to **skip**. We say that a trace $\mathcal{K}$ is *sequentialized with respect to* t if every transition over thread t is in $\Longmapsto_{\mathrm{s}}$. We denote this transition relation by $\Longmapsto_{\mathrm{s}(t)}$.

10. PROOF OF BEHAVIOR PRESERVATION

We now prove the soundness of our analysis, that is, that our parallelizing transformation does not introduce any new behaviors to the original program. We prove our result

using the semantics from Section 9 for both the sequential and the parallelized program. In this setting, the sequential program is represented with a sequential thread[6] possibly executing concurrently with other sequential threads—this degenerates into the purely sequential case when there are no other runnable threads.

10.1. Informal Description of the Proof

The proof of the behavior preservation (Theorem 10.12) is based on a simulation argument: each trace of the parallelized program must correspond to a trace of the sequential program and preserve as an invariant a given binary relation between states of the parallelized and the sequential program. The proof is structured as follows.

(1) We establish a simulation invariant between the parallelized program and the sequential program in two steps: first for a program decorated with calls to **grant**, **wait** and **newch** and a program with these calls erased (Lemma 10.1), and then for a program additionally decorated with **fork** calls and a **fork**-erased program (Lemma 10.2). The combined invariant relates every nonfaulting sequentialized trace of the fully decorated program with a trace of the sequential program. By *sequentialized*, we mean that forked children must execute to completion before their parent threads can be scheduled (Definition 9.6).
(2) We show that, under the assumption that forked child threads never wait for channels granted by their parent or later-forked child threads, any terminating nonfaulting trace of the parallelized program can be reordered into a sequentialized trace with the same thread-local behavior. This is established by showing that traces of the program parallelized by our analysis have signaling barriers aligned with the thread ordering (Lemma 10.8), and that any such trace can be reordered into a sequentialized trace (Lemma 10.7).
(3) Previous steps establish behavior-preservation for nonaborting traces. Inserting **grant**, **wait**, **newch**, and **fork** may introduce aborts. However, our parallelizing transformation is proof-preserving so the parallelized programs is also verified using separation logic. This establishes, for every state satisfying the program precondition, that the program cannot abort.

10.2. Details of the Proof

We start with a lemma that establishes a simulation invariant between a program decorated with calls to **grant**, **wait**, and **newch** and a program with these calls replaced with **skip**.

LEMMA 10.1 (**wait**, **grant**, **newch** INSERTION).
Let $[\![-]\!]^{\downarrow}_{\mathrm{ch}}$ *be a program transformation which replaces calls to* **newch**, **grant**, *and* **wait** *with* **skip**. *Let* $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ *and* κ' *be global states. We say that the invariant* $I(t, \kappa, \kappa')$ *is satisfied if:*

— *The thread identifier* t *is in* $\mathrm{dom}(\delta)$.
— *For* $\delta(t) = \langle C, \sigma, \omega, \gamma \rangle$, *the program* C *executed by* t *does not contain* **fork**.
— *The thread waits for all channels it grants on, that is,* $\omega_{\mathrm{g}} \subseteq \omega_{\mathrm{w}}$, *and all channels in* $\omega_{\mathrm{w}} \backslash \omega_{\mathrm{g}}$ *have already been granted on but are not yet finalized, that is,* $\forall_{c \in \omega_{\mathrm{w}} \backslash \omega_{\mathrm{g}}}. \eta(c) \notin \{\triangledown, \triangle\}$.

[6]A thread is sequential if it does not call `fork`.

— *Let* $\sigma_{\mathrm{ch}} \triangleq \circledast_{c \in \omega_{\mathrm{w}} \setminus \omega_{\mathrm{g}}} . \eta(c)$ *be all resources the thread does not grant but waits for. Then there exists* $\eta' \subseteq \eta$ *such that* $\kappa' = \langle \delta[t \mapsto \langle [\![C]\!]^{\downarrow}_{\mathrm{ch}}, \sigma \oplus \sigma_{\mathrm{ch}}, (\emptyset, \emptyset), \gamma \rangle], \eta', \mathcal{L} \rangle$ *and for all channels* c *held by threads other than* t, $\eta'(c) = \eta(c)$.

For any states κ, κ', κ_2, *if* $I(t, \kappa, \kappa')$ *and* $\kappa \Longrightarrow \kappa_2$, *then there exists a state* κ_2' *such that* $\kappa' \Longrightarrow^* \kappa_2'$ *and* $I(t, \kappa_2, \kappa_2')$. *This property can be alternatively represented by the following diagram:*

$$\begin{array}{ccc} \kappa & \xrightarrow{I(t)} & \kappa' \\ \Downarrow & & \Downarrow{\scriptstyle *} \\ \kappa_2 & \xrightarrow{I(t)} & \kappa_2' \end{array}$$

Intuitively, the invariant I *expresses the simulation relation between programs containing* **wait**, **grant**, *and* **newch** *(left-hand side of the diagram), and the equivalent program without them (right-hand side).*

PROOF. In Appendix A.1. □

The next lemma establishes a simulation invariant between a program containing calls to **fork** and a program with these calls replaced with **skip**.

LEMMA 10.2 (**fork** INSERTION). *Let* $[\![-]\!]^{\downarrow}_{\mathrm{fk}}$ *be a program transformation which replaces calls to* **fork** *with* **skip**. *We say that the invariant* $J(t, \kappa, \kappa')$ *is satisfied if:*

— *The thread identifier* t *is in* $\mathrm{dom}(\delta)$.
— *For* $\delta(t) = \langle C, \sigma, \omega, \gamma \rangle$, *any calls to* **fork** *in* C *do not themselves include calls to* **fork**.
— *Let* $\sigma_{\mathrm{fk}} = \circledast \{\sigma_c \mid c \in \mathrm{child}(t) \wedge \delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle\}$ *be thread-local states and* $\omega_{\mathrm{fk}} = \biguplus \{\omega_c \mid c \in \mathrm{child}(t) \wedge \delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle\}$ *channel states of forked child threads conjoined. By the well-formedness condition on global states (Assumption 9.2), both of these must be well-defined.*
— *There exists at most one child thread* $c \in \mathrm{child}(t)$ *such that* $\delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle$ *and* $C_c \neq$ **skip**.
— *Either of the following two cases holds:*
  — *If* $\forall c \in \mathrm{child}(t) . \delta(c) = \langle$ **skip**$, \sigma_c, \omega_c \rangle$, *then* $\kappa' = \langle \delta[t \mapsto \langle [\![C]\!]^{\downarrow}_{\mathrm{fk}}, \sigma \oplus \sigma_{\mathrm{fk}}, \omega \uplus \omega_{\mathrm{fk}} \rangle], \eta, \mathcal{L} \rangle$.
  (In other words, all child threads of t *have terminated, and the main thread corresponds to the original sequential state.)*
  — *If* $\exists! c \in \mathrm{child}(t) . \delta(c) = \langle C_c, \sigma_c, \omega_c, \gamma_c \rangle \wedge C_c \neq$ **skip**, *then* $C =$ **skip**$; C_1$ *and* $\kappa' = \langle \delta[t \mapsto \langle C_c; [\![C_1]\!]^{\downarrow}_{\mathrm{fk}}, \sigma \oplus \sigma_{\mathrm{fk}}, \omega \uplus \omega_{\mathrm{fk}}, \emptyset \rangle], \eta, \mathcal{L} \rangle$.
  (In other words, there exists exactly one unterminated child thread, and join of the states of the child threads and main thread correspond to the original sequential state.)

For any states κ, κ', κ_2, *if* $J(t, \kappa, \kappa')$, *and* $\kappa \Longrightarrow_{\mathrm{s}(t)} \kappa_2$, *then there exists a state* κ_2' *such that* $\kappa' \Longrightarrow^* \kappa_2'$ *and* $J(t, \kappa_2, \kappa_2')$. *This property can be alternatively represented by the following diagram:*

$$\begin{array}{ccc} \kappa & \xrightarrow{J(t)} & \kappa' \\ \Downarrow{\scriptstyle \mathrm{s}(t)} & & \Downarrow{\scriptstyle *} \\ \kappa_2 & \xrightarrow{J(t)} & \kappa_2' \end{array}$$

The invariant J expresses the simulation relation between programs containing **fork** *(left-hand side of the diagram) and the equivalent program without it (right-hand side).*

PROOF. In Appendix A.2. □

The previous two lemmas combined establish an invariant relating nonfaulting sequentialized traces of the parallelized program with traces of the sequential program. It remains to show that terminating nonfaulting traces of the parallelized program can be reordered into sequentialized traces with the same thread-local behavior. We do this by introducing a well-founded order on traces in which sequentialized traces are minimal elements, and showing that each signal-ordered trace (i.e., having its signaling barriers aligned with the thread ordering) can be reordered to a behaviorally equivalent trace smaller with respect to the well-founded order, eventually leading to a behaviorally equivalent sequentialized trace.

Definition 10.3. We say that traces $\mathcal{K}_1$ and $\mathcal{K}_2$ are *behaviorally equivalent* if, for each thread identifier t, the sequence of transitions over t in $\mathcal{K}_1$ and $\mathcal{K}_2$ are identical.

We first show that consecutive transitions of two different threads can be reordered if the second thread does not wait for any channel that the first thread grants on. We use this property in the subsequent lemma to inductively construct a behaviorally equivalent sequentialized trace.

LEMMA 10.4 (TRACE REORDERING). *Suppose we have a two-step trace:*

$$\mathcal{K} \;=\; \langle \delta_1, \eta_1, \mathcal{L}_1\rangle \overset{t_1}{\Longmapsto} \langle \delta_2, \eta_2, \mathcal{L}_2\rangle \overset{t_2}{\Longmapsto} \langle \delta_3, \eta_3, \mathcal{L}_3\rangle$$

such that $t_1 \neq t_2$, and that $t_1, t_2 \in \mathrm{dom}(\delta_1)$. Let $\delta_1(t_1) = \langle C, \sigma, \omega, \gamma\rangle$ and $\delta_1(t_2) = \langle C', \sigma', \omega', \gamma'\rangle$. Suppose that t_2 does not wait for any channel that t_1 grants on, that is, that $\omega_{\mathrm{g}} \cap \omega'_{\mathrm{w}} = \emptyset$ holds. Then there exists a thread environment η', an unallocated set $\mathcal{L}'$, and a two-step trace:

$$\mathcal{K}' \;=\; \langle \delta_1, \eta_1, \mathcal{L}_1\rangle \overset{t_2}{\Longmapsto} \langle \delta_1[t_2 \mapsto \delta_3(t_2)], \eta', \mathcal{L}'\rangle \overset{t_1}{\Longmapsto} \langle \delta_3, \eta_3, \mathcal{L}_3\rangle$$

The new trace $\mathcal{K}'$ is behaviorally equivalent (Definition 10.3) to $\mathcal{K}$.

PROOF. In Appendix A.3. □

Definition 10.5 (*signal order*). We describe a trace $\mathcal{K}$ as *signal-ordered with respect to t* if t is a thread identifier such that for all states $\kappa = \langle \delta, \eta, \mathcal{L}\rangle \in \mathcal{K}$:

$$\forall c \in \mathrm{child}(\kappa, t).\, \forall k \in \mathrm{waits}(\kappa, c).$$
$$((\exists c' \in \mathrm{child}(\kappa, t).\, c' < c \wedge k \in \mathrm{grants}(\kappa, c')) \;\vee\; (\eta(k) \notin \{\bot, \triangledown, \triangle\}))$$

(In other words, if a child of t can wait on a channel k, some earlier child must be able to grant on it, or the channel must already be filled.)

Definition 10.6 (*degree of sequentialization*). Let $\mathcal{K}$ be a trace. If $\mathcal{K}$ is not sequentialized with respect to t, let $c \in \mathrm{child}(t)$ be the earliest thread in the child order with an unsequentialized transition (that is, for all other children c' with unsequentialized transitions, $c' > c$). Now, let F be the number of transitions from the originating **fork** transition for c and the end of the trace. Let K be the number of transitions over c *after* the first unsequentialized transition. Let D be the number of intervening transitions between the originating **fork** and the first unsequentialized transition.

We call such a tuple (F, K, D) the *degree of sequentialization* with respect to t. We order such tuples lexicographically, with left-to-right priority, and lift this order to give a partial order on traces $<_t$. This order on traces is well-founded, with sequentialized traces occupying the minimum position in the order.

Using the ordering on traces introduced in Definition 10.6 we now show that a trace that is signal-ordered with respect to a thread t can be reordered to a behaviorally equivalent trace that is sequentialized with respect to t.

LEMMA 10.7 (SEQUENTIALIZATION). *Let $\mathcal{K}$ be a trace such that*

— *$\mathcal{K}$ is signal-ordered with respect to t; and*
— *Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be the final state in $\mathcal{K}$ and let t' be the maximum thread in* $\mathrm{child}(\kappa, t)$ *with transitions in $\mathcal{K}$. For all threads $t'' \in \mathrm{child}(\kappa, t)$ such that $t'' < t'$, $.\delta(t'') = \langle \mathbf{skip}, \sigma, \omega, \gamma \rangle$ for some σ, ω, γ.* (Note this holds automatically if $\mathcal{K}$ is terminating.)

Then there must exist a trace $\mathcal{K}'$ sequentialized with respect to t such that $\mathcal{K}$ is behaviorally equivalent to $\mathcal{K}'$.

PROOF. In Appendix A.4. □

We now establish that the pattern of barriers that we insert into the program during our parallelization analysis results in a trace that is signal-ordered. Recall from Section 5 that the parallelized program constructed by our analysis has the following form[7]:

```
ci := newch();
grant(ci);
C1;
while(B) {
  C2;
  cj := newch();
  fork[P](C3);
  ci := cj;
}
wait(ci);
```

Here we assume that we need only a single active pair of channels `ci`/`cj`—the argument generalizes straightforwardly to the case with n channels. In addition, we assume the subprograms `C1` and `C2` do not include `wait`, `grant`, `newch` and `fork`, and that they do not assign to the variables `ci` and `cj`.

The assertion P is defined so that: $P \vdash \mathsf{fut}(\mathtt{ci}, P_1) * \mathsf{req}(\mathtt{cj}, P_2) * F$, for some P_1, P_2 and F; and $F \nvdash \mathsf{fut}(x, P')$ and $F \nvdash \mathsf{req}(x, P')$ for any x and P'. (In other words, the forked thread can only wait for `ci` and grant on `cj`.)

LEMMA 10.8 (CONSTRUCTION OF SIGNAL-ORDERED TRACES). *Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be a state and t a thread identifier such that $\delta(t) = \langle C, \sigma, (\emptyset, \emptyset), \emptyset \rangle$ and C matches the standard form for parallelized programs, described above. Let $\mathcal{K}$ be a trace with initial state κ. Then $\mathcal{K}$ is signal-ordered with respect to t.*

PROOF. In Appendix A.5. □

In Lemma 10.8 we assumed a specific program form constructed by the parallelizing transformation from Section 5.2. However, the soundness proof could be adapted to hold for other parallelization backends. As long as the analysis generates a parallelized program with traces that are signal-ordered we could establish an analogue of Lemma 10.8 and proceed with other steps of the soundness proof in the same way.

LEMMA 10.9 (SYNCHRONIZATION ERASURE). *Let C be a sequential program, and C_{par} be a program resulting from applying our parallelization analysis. Then*

[7]Recall also that we define **for** in terms of **while**.

$C = [\![[\![C_{\text{par}}]\!]^{\downarrow}_{\text{fk}}]\!]^{\downarrow}_{\text{ch}}$, *where the (syntactic) equality of programs is defined up to insertion/erasure of* **skip** *statements—that is syntactic manipulation using the identity* $C = C; \textbf{skip}$.

PROOF. Our analysis only inserts channels and calls to fork. The result follows trivially. □

LEMMA 10.10 (MATERIALIZED VARIABLES). *Materializing local variables, as described in Section 3.2, has no effect on the operational behavior of a program.*

PROOF. Let C be a program, and let C' be a corresponding program in which the local variables are materialized. Let $\mathcal{K}$ be a trace in which the program C is executed. When inserted, the materialized variables are written but not read (they are *subsequently* used in conditionals after loop-splitting). Therefore, there must be a trace $\mathcal{K}'$ in which C' is substituted for C, which is identical to $\mathcal{K}$ aside from the materialized variables. □

LEMMA 10.11 (LOOP-SPLITTING). *Loop-splitting, as described in Section 5.2, has no effect on the operational behavior of the program.*

PROOF. By the fact that either a conditional or its negation must hold, and a straightforward appeal to the semantics of loops. □

Finally, we prove the main theorem stating the soundness of our parallelization analysis.

THEOREM 10.12 (PARALLELIZATION SOUNDNESS). *Let C be a sequential program, and let C_{par} be the program resulting from applying our parallelization analysis to C (including variable materialization and loop-splitting). Let $\mathcal{K}$ be a nonfaulting terminating trace with initial state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$, and let t be a thread identifier such that $\delta(t) = \langle C_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$. Then there exists a trace $\mathcal{K}''$ with initial state $\kappa'' = \langle \delta[t \mapsto \langle C, \sigma, (\emptyset, \emptyset), \emptyset \rangle], \eta, \mathcal{L} \rangle$ such that:*

(1) *For all thread identifiers t' defined in $\mathcal{K}''$ such that $t \neq t'$, their thread-local behavior in $\mathcal{K}$ is identical to $\mathcal{K}''$.*
(2) *Let κ_Ω be the final state in $\mathcal{K}$, and κ''_Ω the final state of $\mathcal{K}''$. Let σ be the local state associated with thread t in κ_Ω and σ'' the corresponding state associated with t in κ''. There exists a state σ' such that $\sigma \oplus \sigma' = \sigma''$.*

PROOF. By Lemma 10.8 the trace $\mathcal{K}$ must be signal-ordered with respect to t. Therefore by Lemma 10.7 there exists a behaviorally equivalent trace $\mathcal{K}_{\text{seq}}$ that is sequentialized with respect to t. Because t has no child threads in κ, we can choose $\mathcal{K}_{\text{seq}}$ such that κ is also its initial state.

We now show that there exist traces $\mathcal{K}'$ and $\mathcal{K}''$, such that transitions in $\mathcal{K}$ are related to $\mathcal{K}'$ by the invariant $J(t)$ (Lemma 10.2) and transitions in $\mathcal{K}'$ are related to $\mathcal{K}''$ by the invariant $I(t)$ (Lemma 10.1).

We proceed by induction on the length of a prefix of $\mathcal{K}_{\text{seq}}$. By Lemma 10.9, $C = [\![[\![C_{\text{par}}]\!]^{\downarrow}_{\text{fk}}]\!]^{\downarrow}_{\text{ch}}$. From this, it is straightforward to see that there exists a state $\kappa' = \langle \delta[t \mapsto \langle [\![C_{\text{par}}]\!]^{\downarrow}_{\text{fk}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle], \eta, \mathcal{L} \rangle$ such that $J(t, \kappa, \kappa')$ and $I(t, \kappa', \kappa'')$.

Assume traces $\mathcal{K}'$ and $\mathcal{K}''$ exist for the first n transitions of $\mathcal{K}_{\text{seq}}$. Let κ_n be the final state in this prefix. By assumption there exist states κ'_n and κ''_n such that $J(t, \kappa_n, \kappa'_n)$ and $I(t, \kappa'_n, \kappa''_n)$. Now show that for the next transition $\kappa_n \Longmapsto_{\text{s}(t)} \kappa_{n+1}$ there exist transitions

$\kappa'_n \Longmapsto^* \kappa'_{n+1}$ and $\kappa''_n \Longmapsto^* \kappa''_{n+1}$ such that $J(t, \kappa_{n+1}, \kappa'_{n+1})$ and $I(t, \kappa'_{n+1}, \kappa''_{n+1})$. This can be visualized by the following diagram:

$$\begin{array}{ccccc} \kappa_n & \xrightarrow{J(t)} & \kappa'_n & \xrightarrow{I(t)} & \kappa''_n \\ \Big\Downarrow{\scriptstyle s(t)} & & \Big\Downarrow{\scriptstyle *} & & \Big\Downarrow{\scriptstyle *} \\ \kappa_{n+1} & \xrightarrow{J(t)} & \kappa'_{n+1} & \xrightarrow{I(t)} & \kappa''_{n+1} \end{array}$$

This is an immediate consequence of Lemmas 10.1 and 10.2.

By induction, this suffices to establish the existence of the trace $\mathcal{K}''$. Corresponding thread-local behavior for threads $t' \neq t$, and corresponding final state for thread t follow immediately from the definitions of $I(t)$ and $J(t)$.

This result also holds if our analysis performs variable materialization and loop-splitting; to show this, we need only appeal to Lemma 10.10 and Lemma 10.11. □

COROLLARY 10.13. *If the chosen postcondition in the proof of the program is precise, then the result of parallelization is an equality, not a substate.*

PROOF. Consequence of the fact that at most one substate of a given state can satisfy a precise assertion. □

11. TERMINATION

In this section we prove that our analysis cannot introduce nontermination to the parallelized program.

LEMMA 11.1 (ENSURING GRANT). *Let C_{par} be a program resulting from applying our parallelization analysis to some sequential program C. Let $\mathcal{K}$ be a trace with initial state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$, such that $\delta(t) = \langle C_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$. For all states in $\kappa' = \langle \delta, \eta, \mathcal{L} \rangle \in \mathcal{K}$ and all children $c \in \text{child}(\kappa', t)$ if c has terminated in κ' (i.e., reduced to* **skip***) then* $\text{grants}(\kappa', c) = \emptyset$.

PROOF. Consequence of the fact that in our analysis, each **grant** is injected along every control-flow path. Consequently, a terminating thread must call **grant** for every channel to which it has access. □

LEMMA 11.2 (TRACE EXTENSION). *Let $\mathcal{K}$ be a trace that is signal-ordered with respect to t, and let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be the final state of $\mathcal{K}$. Let t' be the minimum child of t such that $\delta(t') = \langle C, \sigma, \omega, \gamma, \rangle$ and $C \neq$* **skip**. *Then there exists a nonfaulting transition $\kappa \overset{t'}{\Longmapsto} \kappa'$.*

PROOF. By examination of the thread-local semantics, we can see that only calls to **wait** can block without faulting. For all other commands, either the thread can take a step, or it faults immediately. Consequently, the lemma reduces to asking whether every call to **wait** in the thread t' can take a step.

wait can only block if some prior thread has not called **grant** on some channel. By the definition of signal-ordered (Definition 10.5), the channels for which t' can wait must either contain state, or must be held by some child thread earlier in the order. By assumption, we know that all earlier threads have terminated, so by Lemma 11.1, we know that no thread holds a pending grant. Consequently, the channel must contain state, and **wait** can take a step. □

LEMMA 11.3 (TRACE ERASURE). *Let $\mathcal{K}$ be a trace that is signal-ordered with respect to t, and let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ be the final state of $\mathcal{K}$. Let t' be the maximum child of t in* $\text{dom}(\delta)$.

Then there exists a trace $\mathcal{K}'$ such that $\mathcal{K}'$ has no transitions over t', and all threads other than t' have identical behavior in $\mathcal{K}$ and $\mathcal{K}'$.

PROOF. By induction over the number of transitions over t in the trace. identify the final transition over t' in $\mathcal{K}$, $\kappa_n \overset{t'}{\Longmapsto} \kappa_{n+1}$. We erase this transition, and then reexecute the remainder of the transitions from the trace. By the definition of signal-order (Definition 10.5) no other child thread can call **wait** on any channel held by t. Consequently, we must be able to run the same sequence of thread-local actions after erasing the transition. By applying this process, we can erase all transitions over t'. □

THEOREM 11.4 (TERMINATION). *Let C be a sequential program, and let C_{par} be a program resulting from applying our parallelization analysis to C. If C is guaranteed to terminate, then so is C_{par}.*

PROOF. If C is terminating, then for any initial state, there must be a maximum number of steps that the thread t running C can take. To prove the theorem, we prove the contrapositive result: given that C_{par} does not terminate, we can construct a trace for C_{seq} where t takes more than this number of steps.

Suppose we have an initial state $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ such that $\delta(t) = \langle C_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle$. Suppose that κ can result in a nonterminating trace over t—that is, given any arbitrary bound n, there exists a trace with initial state κ and more than n steps over t and its child threads.

Construct a sequential state $\kappa_{\text{seq}} = \langle \delta[t \mapsto \langle C_{\text{par}}, \sigma, (\emptyset, \emptyset), \emptyset \rangle], \eta, \mathcal{L} \rangle$, and identify a bound q such that no trace starting with κ'' can take more than q steps over t. Choose a trace $\mathcal{K}$ with initial state κ such that t and its children take more than q steps, *excluding* synchronization (that is, calls to **fork**, **newch**, **signal**, and **wait**). Note that by Lemma 10.8, $\mathcal{K}$ is signal-ordered.

Now we rearrange $\mathcal{K}$ to give a sequentialized trace of length greater than q.

— By applying Lemma 11.2, we extend the trace until the first n children of t together take more than q nonsynchronization steps, and all children but the nth have terminated.
— By applying Lemma 11.3, we erase the $(n+1)$th to maximum child threads of t, along with associated calls to fork from t.

The resulting trace $\mathcal{K}$ has more than q nonsynchronization transitions from t and its first n children, and in the final state κ', all but the nth child thread have terminated. By applying Lemma 10.7, we can construct a trace $\mathcal{K}''$ that has equivalent thread-local behavior, and that is sequentialized with respect to t.

We observe that, by Lemma 10.9, $C = [\![[\![C_{\text{par}}]\!]^{\downarrow}_{\text{fk}}]\!]^{\downarrow}_{\text{ch}}$. By applying Lemmas 10.1 and 10.2, construct a trace $\mathcal{K}_{\text{seq}}$ with initial state κ_{seq}. The invariants in Lemmas 10.1 and 10.2 only erase synchronization transitions. Consequently, the resulting sequential trace has more than q transitions over the thread t. This contradicts our assumption, and completes the proof. □

12. RELATIONSHIP TO OTHER APPROACHES

Traditional approaches to automatic parallelization are generally based on checking the independence of program statements. Here, a compiler performs dependency analysis, generating constraints about program statements, and schedules statements that are independent of each other for parallel execution. Dependency constraints are typically derived from a program analysis that discovers how the result of one computation affects the result of another.

These techniques have been successful for programs with simple data types, statically allocated arrays, and structured control-flow operators, such as loops, but have been less effective when programs manipulate pointers and make extensive use of dynamic data structures [Ghiya et al. 1998; Gupta et al. 1999; Hendren and Nicolau 1990; Horwitz et al. 1989]. In our work, we leverage separation logic because it provides a convenient means to express assertions about dynamic or recursively defined resources. For example, Raza et al. [2009] investigate how a separation logic based analysis can be adapted to express memory separation properties, thereby detecting independence between statements via interference checking of proof assertions. In our approach, we do not check for such independence directly in the proof, but instead use the proof to discover what resources different parts of the program depend on, and inject appropriate synchronization to ensure that the parallelized version of the program respects these dependencies.

Thus, there are three main differences between our proposed technique and prior work.

— In contrast to other approaches where the proof produced by a compiler is of special form, strictly targeted at checking independence, we do not make any assumptions about the program proof except that it be written in separation logic. By virtue of separation logic's semantics, any such proof describes all resources accessed by the program—a property that we crucially rely on to track fine-grained sequential dependencies.
— Our approach starts with an unconstrained parallelized version of the program, and then inserts just enough synchronization to preserve sequential dependencies and guarantee preservation of observable behavior. By releasing the resources that a thread does not need and acquiring the resources that it does, we implicitly establish (an over-approximated) independence of certain parts of the program as a byproduct of our analysis.
— While the main usage scenario that we have presented is do-across parallelization of for-loops with *heap-carried* dependencies, our key concepts are in fact not dependent on a specific loop structure. Our approach is equally applicable to less regular code patterns and other iteration structures.

12.1. Related Work in Detail

Resource-usage inference by abduction. We have defined an inter-procedural, control-flow-sensitive analysis capable of determining the resource that will (and will not) be accessed between particular points in the program. At its core, our analysis uses abductive reasoning [Calcagno et al. 2011] to discover redundancies, that is, state used earlier in the program that will not be accessed subsequent to the current program point. Using abduction in this way was first proposed in [Distefano and Filipović 2010], where it is used to discover memory leaks, albeit without conditionals, procedures, loops, or code specialization.

In Calcagno et al. [2009], abduction is used to infer resource invariants for synchronization, using a process of counterexample-driven refinement. Our approach similarly infers resource invariants, but using a very different technique: invariants are derived from a sequential proof, and we also infer synchronization points and specialize the program to reveal synchronization opportunities.

Behavior-preserving parallelization. We expect our resource-usage analysis can be used in other synchronization-related optimizations, but in this article, we have used it as the basis for a parallelizing transformation. This transformation is in the style of *deterministic parallelism* [Bergan et al. 2010; Berger et al. 2009; Bocchino et al. 2009; Burnim and Sen 2010]—although our approach does not, in fact, require determinacy

of the source program. In this vein, our transformation ensures that every behavior of the parallelized program is a behavior of the source sequential program (modulo the caveats about allocation and termination discussed in Section 6).

Previous approaches to deterministic parallelism have not used specifications to represent dynamically allocated resources. This places a substantial burden on the analysis and runtime to safely extract information on resource usage and transfer—information that is readily available in a proof. As a result, these analyses tend to be much more conservative in their treatment of mutable data. Our proof-based technique gives us a general approach to splitting mutable resources; for example, by allowing the analysis to perform ad-hoc list splitting, as we do with `move()`.

Loop parallelization. Our approach transforms a sequential `for`-loop by running all the iterations in parallel and signaling between them, so it can be seen as a variant of a do-across style of loop parallelization [Tang et al. 1994]. There has been a vast amount of work in last couple of decades on do-across dependence analysis and parallelization for loop nests. However, the focus of interest of these approaches are loop nests over linear data structures (i.e., arrays). For instance, polyhedral models [Baskaran et al. 2009], the most powerful dependence abstraction to date, assume affine iteration domain of loop nests. In contrast, our approach does not require specific shape of the iteration domain or fixed data-structure access patterns.

Instead of parallel-`for`, we could have used a more irregular concurrency annotation, for example safe futures [Navabi et al. 2008], or an unordered parallel-`for`, as in the Galois system [Pingali et al. 2011]. In the former case, our resource-usage analysis would be mostly unchanged, but our parallelized program would construct a set of syntactically distinct threads, rather than a pipeline of identical threads. Removing ordering between iterations, as in the latter case, would require replacing ordered `grant-wait` pairs with, for instance, conventional locks, and would then introduce an obligation to show that locks were always acquired together, as a set.

Proof-driven parallelization and program analyses. A central insight of our approach is that a separation logic proof expresses data dependencies for parts of a program, as well as for the whole program. These internal dependencies can be used to inject safe parallelism. This insight is due to Raza et al. [2009], Cook et al. [2010], and Hurlin [2009], both of which propose parallelization analyses based on separation logic. The analyses proposed in these papers are much more conservative than ours, in that they discover independence which already exists between commands of the program. They do not insert synchronization constructs, and consequently cannot enforce sequential dependencies among concurrent computations that share and modify state. Indeed, Raza et al. [2009] do not consider any program transformations, since the goal of that work is to identify memory separation of different commands, while Hurlin [2009] expresses optimizations as reordering rewrites on proof trees.

Bell et al. [2009] construct a proof of an already-transformed multithreaded program parallelized by the DSWP transformation [Ottoni et al. 2005]. This approach assumes a specific pattern of (linear) dependencies in the while-loop consistent with DSWP, a specific pattern of sequential proof, and a fixed number of threads. In our `move()` example, the outermost (parallelizing) loop contains two successive inner loops, while the example in Figure 1 illustrates how the technique can deal with inter-procedural and control-flow sensitive dependencies. In both cases, the resulting parallelization is specialized to inject synchronization primitives to enforce sequential dependencies. We believe examples like these do not fall within the scope of either DSWP or the proof techniques supported by Bell et al. [2009].

Outside separation logic, Deshmukh et al. [2010] propose an analysis which augments a sequential library with synchronization. This approach takes as input

a sequential proof expressing the correctness criteria for the library, and generates synchronization ensuring this proof is preserved if methods run concurrently. A basic assumption is that the sequential proof represents all the properties that must be preserved. In contrast, we also preserve sequential order on access to resources. Consequently, Deshmukh et al. permit parallelizations that we would prohibit, and can introduce new behaviors into the parallelized program. Another difference is that [Deshmukh et al. 2010] derive a linearizable implementation given a sequential specification in the form of input-output assertions; because they do not consider specialization of multiple instances of the library running concurrently, it is unclear how their approach would deal with transformations of the kind we use for `move()`.

Golan-Gueta et al. [2011] also explore adding locks to sequential libraries; in particular, they focus on tree- and forest-based data structures. They instrument the library's code so that it counts stack and heap references to objects, and then use these reference counts to determine when to acquire and release locks to guarantee conflict-serializability (and consequently, linearizability). The use of proofs in Golan-Gueta et al. [2011] is, however, only indirect: they use a shape analyzer to check that the library's heap graph is tree-shaped and live variable analysis to eliminate redundant code. Another difference is that our analysis does not assume a specific heap structure at any single point in the program nor it requires additional instrumentation of heap objects.

Raychev et al. [2013] describe an alternative program analysis and synchronization inference mechanism. They take as an input a nondeterministic program (obtained by, for instance, "naively" parallelizing the sequential program) and make it deterministic by inserting barriers that enforce a thread schedule that avoids races. The thread ordering is determined by eliminating conflicts in a thread-modular abstraction of the program, following a set of rules to decide between feasible schedules. As a consequence, Raychev *et al.* end up with a determinization whose behavior is implied by these rules rather than by the behavior of the original sequential program. To abstract the unbounded state, they use a flow-insensitive pointer analysis and numerical abstract domains, which works well for the structured numerical computations over arrays analyzed in their examples, but which would be much less effective for more irregular computations over dynamic data structures, as we consider here. For convergence, they employ a simple widening strategy (they widen after a constant number of iterations around the loop), which does not allow splitting of loop invariants, as we do in the transformation of `move()`.

Separation logic and concurrency. Separation logic is essential to our approach. It allows us to localize the behavior of a program fragment to precisely the resources it accesses. Our proofs are written using concurrent separation logic [O'Hearn 2007]. CSL has been extended to deal with dynamically allocated locks [Gotsman et al. 2007; Hobor et al. 2008; Jacobs and Piessens 2009], reentrant locks [Haack et al. 2008], and primitive channels [Bell et al. 2009; Hoare and O'Hearn 2008; Leino et al. 2010; Villard et al. 2010]. Sequential tools for separation logic have achieved impressive scalability—for example Calcagno et al. [2011] have verified a large proportion of the Linux kernel. Our work can be seen as an attempt to leverage the success of such sequential tools. Our experiments are built on coreStar [Botinčan et al. 2011; Distefano and Parkinson 2008], a language-independent proof tool for separation logic.

The parallelization phase of our analysis makes use of the specifications for parallelization barriers proposed in Dodds et al. [2011]. That paper defined high-level specifications representing the abstract behavior of barriers, and verified those specifications against the barriers' low-level implementations. However, it assumed that barriers were already placed in the program, and made no attempt to infer barrier

positions. In contrast, we assume the high-level specification, and define an analysis to insert barriers. The semantics of barriers used in that paper and here was initially proposed in Navabi et al. [2008].

Deterministic Parallelism. As questions regarding the programmability of multicore systems become increasingly vocal, deterministic parallelism offers an attractive simplified programming model that nonetheless yield performance gains. Safe futures [Welc et al. 2005] provide deterministic guarantees for Java programs annotated with a `future` construct using software transactional memory infrastructure; Grace [Berger et al. 2009] is another runtime technique that uses memory protection hardware for the same purpose. In both these systems, the observable behavior of the parallelized program is the same as a sequential version, a version in which future creation operations are treated as no-ops.

Deterministic parallelism can also be profitably applied to explicitly parallel applications; these systems guarantee that the program produces the same output regardless of the order in which threads run, but this result need not necessarily match a sequential execution. CoreDet [Bergan et al. 2010], is a compiler and runtime system that uses ownership information along with a versioned memory to commit changes deterministically for multithreaded C++ programs. Dthreads [Liu et al. 2011] works at a higher level as a direct replacement for the `pthreads` library, improving the scalability with an efficient commit protocol and amortization of overheads. Deterministic Parallel Java [Bocchino et al. 2009] uses a type and region effect system to enforce deterministic guarantees for concurrent Java programs.

ELECTRONIC APPENDIX

The electronic appendix to this article is available in the ACM Digital Library.

APPENDIXES

A.1. Proof of Lemma 10.1

PROOF. There are two cases: either $\kappa \Longmapsto \kappa_2$ is a transition over t, or it is a transition over some other thread $t' \neq t$. If $\kappa \Longmapsto \kappa_2$ is a transition over t, we proceed by structural induction over C.

- C is a primitive command in Prim. In this case, the result holds by the assumption of behavioral monotonicity for primitive commands.
- C is a **grant** or **wait** for some channel c. In both of these cases, the corresponding command $[\![C]\!]^{\downarrow}_{\mathrm{ch}}$ will be **skip**. Consequently the transition $\kappa' \Longmapsto \kappa'_2$ will be a τ-transition, meaning $\kappa' = \kappa'_2$.
  By assumption, the join of the thread-local state σ and state σ_{ch} stored in accessible channels in κ is equal to the thread-local state in κ'. Calling a **grant** takes some local state and pushes it into a channel, which preserves this property. Similarly calling a **wait** pulls some state out of an accessible channel and into local state, which also preserves the property.
- C is a **newch**. In this case, the corresponding command also be a **skip**, so $\kappa' = \kappa'_2$. The effect of **newch** will be to initialize a fresh channel not already in η, and add these channels to the channel state for κ_2. These channels are erased by the invariant, so the property is preserved.
- C is **alloc**. The set $\mathcal{L}$ of unallocated locations is the same in κ and κ'. Consequently, if a location can be allocated in κ, it can also be allocated in κ'.
- C is a sequential composition, loop, or conditional. The result follows trivially by appeal to the induction hypothesis.

Suppose now that the transition is over some other thread $t' \neq t$. By assumption (the third property of I), we know that the channels erased by the invariant are not shared with any other thread. We can therefore establish straightforwardly that such a transition must be replicated exactly between κ' and κ'_2. The invariant is not disturbed, because it does not mutate the content of other threads. □

A.2. Proof of Lemma 10.2

PROOF. Unlike with Lemma 10.1, there are *three* cases: either $\kappa \Longmapsto_{\mathrm{s}(t)} \kappa_2$ is a transition over t; or it is a transition over some member of the set child(t); or the transition is over some other, unrelated thread. First suppose that the transition is over t. We proceed by structural induction on C.

— $C \in \mathsf{Prim}$. By assumption, any behavior in a small state σ will be reflected in a big one, which suffices to ensure that the invariant holds.
— C is **alloc**. The set $\mathcal{L}$ of unallocated locations is the same for κ and κ'. By the same argument as Lemma 10.1, the same allocation step is therefore possible in both global states.
— C is **wait** or **grant**. By the well-formedness assumption, resources held by channels must be disjoint from those held by other threads. This suffices to ensure that **wait** and **grant** transitions can take place identically in the corresponding **fork**-free program.
— C is **fork**. This will correspond to a τ-transition in the **fork**-erased program. Because the transition over the program with **fork** is sequentialized with respect to t, the transition can only be over t if all the children of t have reduced to **skip**. Calling **fork** pushes some state to the child thread, and results in exactly one active (non-**skip**) child thread.
— C is a loop, conditional, sequential composition. Property holds by appeal to the induction hypothesis.

Now suppose the transition is over some thread $c \in \mathbf{child}(t)$. By assumption (the fourth property of J), there can be at most one such thread. The state held by the child thread must be a substate of that held in the **fork**-free program. Consequently, an almost identical structural induction argument suffices to show that the simulation property holds.

Finally, suppose that the transition is over some other thread t' that is neither t nor a child of t. Such threads cannot be affected by the behavior of t and its children. It is straightforward to show that any transition for the program can take place in the corresponding **fork**-free program. □

A.3. Proof of Lemma 10.4

PROOF. We proceed by case-splitting on the shape of transitions over t_1 and t_2.

— One or both of the transition are thread-local (i.e., do not result in a labeled transition in the thread-local semantics). It is straightforward to see that neither transition can affect the other one, and the rearrangement result follows trivially.
— One or both of the transitions is a **fork**. The effect of such a transition is localized to the parent and newly created child thread. By assumption, neither thread was created by either of these transitions. The rearrangement result follows straightforwardly.
— One or both transitions is a **newch**. Channels can only be transferred by **fork**, and by assumption, neither transition created either of the threads. Consequently the effect of a **newch** call is confined to the local thread.

— Both transitions are **wait** (resp. **grant**). The threads could only affect one another if both waited (resp. granted) on the same channel, but this is ruled out by the well-formedness assumption that each end of each channel is held by at most one thread.
— One transition is a **wait** and the other is a **grant**. Once again, the threads could only affect one another if they waited and granted on the same channel. However, by assumption of the lemma, the thread t_1 cannot grant on any channel on which t_2 can wait. By the structure of the semantics, the transition t_1 cannot wait for a channel that is subsequently granted. So this case cannot occur.
— One or both transitions are a **alloc**. If one operation is a **alloc** and the other another command, the threads trivially cannot affect one another. If both transitions are **alloc**, the locations allocated must be distinct, and the transitions again can be exchanged straightforwardly.

The constructed trace $\mathcal{K}'$ is trivially behaviorally equivalent to $\mathcal{K}$, since the thread-local actions for each thread are identical. □

A.4. Proof of Lemma 10.7

PROOF. We show that any out-of-order trace $\mathcal{K}$ can be reordered by applying Lemma 10.4, to give a behaviorally equivalent trace with a lower degree of sequentialization w.r.t. t.

Consider a trace $\mathcal{K}$ that is signal-ordered but not sequentialized. Let t_2 be the earliest thread in the child order for t that has an unsequentialized transition. By the assumption that the transition is unsequentialized, the preceding transition in the trace must be on some other thread $t_1 \neq t_2$, and must not be the **fork** that created the out-of-order transition. Note that t_1 also must not be some child of t earlier in the child order—otherwise the transition over t_1 would be the earliest out-of-order transition.

By the definition of signal-ordering we know that $\text{wait}(\kappa, t_2) \cap \text{grant}(\kappa, t_1) = \emptyset$. Consequently, we can apply Lemma 10.4, and push the transition over t_2 earlier than the transition over t_1. Call the resulting trace $\mathcal{K}''$.

By Lemma 10.4, $\mathcal{K}''$ is behaviorally equivalent to $\mathcal{K}$. As a trivial consequence, $\mathcal{K}''$ is also signal-ordered. As a deeper consequence, $\mathcal{K}'' <_t \mathcal{K}$ (with respect to the order given in Definition 10.6). There are three cases.

— The transition shifts left, but is still unsequentialized. The number of intervening transitions from the originating **fork** decreases, while the other two measures are unchanged.
— The left shift means the transition is sequentialized, but more transitions over the same thread are unsequentialized. The number of transitions from the first unsequentialized transition decreases, while the distance from the initial **fork** to the end of the thread is unchanged.
— The left shift means all transitions for the thread are sequentialized. The next target thread must be later in the child order, meaning it is later in the trace. The number of transitions from the initial **fork** to the end of the thread decreases.

As the order is well-founded, by repeatedly applying the lemma, we get a behaviorally-equivalent trace that is sequentialized w.r.t. t. □

A.5. Proof of Lemma 10.8

PROOF. We establish this property by defining an invariant $L(\kappa)$.

— Let $\kappa = \langle \delta, \eta, \mathcal{L} \rangle$ and $\delta(t) = \langle C, \sigma, \omega, \gamma \rangle$.
— There exists a channel c_1 such that $\sigma_s(\texttt{ci}) = c_1$, and $\text{waits}(\kappa, t) = \{c_1\}$.
— Either $\eta(c_1) \notin \{\bot, \triangledown, \triangle\}$, or $\exists t' \in \text{child}(\kappa, t).\, c_1 \in \text{grants}(\kappa, t')$.

We now show that signal-ordering holds for every step of the semantics, and that the invariant holds for any state immediately preceding the execution of an iteration of the main `while`-loop; that is, at any point when C = `while(B){ ... }; wait(ci);`.

— Suppose we execute from the beginning of the program (the first command in `C1`) to the beginning of the first iteration **while**. It is simple to see by inspection of the semantics that the resulting state κ will satisfy the invariant L. We create the channel c_1, and the call to **grant** ensures $\eta(c_1)$ is associated with some state, as required by the invariant.
— Now suppose we execute a single iteration of the loop: that is, we assume the boolean condition `B` holds, and apply the appropriate rule in the operational semantics. The invariant cannot be disturbed by other running threads, because by definition they can only call **grant** on the channel c_1.
By assumption, the subprogram `C2` cannot disturb the invariant. We call **newch**, which creates a channel c_2, associated with the variable `cj`. Now we observe that, by the structure of P call to **fork** creates a thread t' with $\text{waits}(\kappa, t') = \{c_1\}$ and $\text{grants}(\kappa, t') = \{c_2\}$. As a result, in the resulting state $\text{waits}(\kappa, t) = \{c_2\}$. The final assignment associates c_2 to `ci`, which reestablishes the invariant.

Each step in this execution satisfies signal-ordering. By induction this completes the proof. □

ACKNOWLEDGMENTS

The authors would like to thank Dino Distefano, Matthew Parkinson, Mohammad Raza, John Wickerson, and the anonymous reviewers for POPL 2012 and TOPLAS for many helpful comments and suggestions.

REFERENCES

Baskaran, M. M., Vydyanathan, N., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. 2009. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 219–228.

Bell, C. J., Appel, A., and Walker, D. 2009. Concurrent separation logic for pipelined parallelization. In *Proceedings of the International Static Analysis Symposium*. 151–166.

Berdine, J., Calcagno, C., and O'Hearn, P. W. 2005a. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*. Springer, 115–137.

Berdine, J., Calcagno, C., and O'Hearn, P. W. 2005b. Symbolic execution with separation logic. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 52–68.

Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. 2010. CoreDet: A compiler and runtime system for deterministic multithreaded execution. *SIGPLAN Not. 45*, 3, 53–64.

Berger, E. D., Yang, T., Liu, T., and Novark, G. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 81–96.

Bocchino, Jr., R. L., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 91–116.

Bornat, R., Calcagno, C., O'Hearn, P., and Parkinson, M. 2005. Permission accounting in separation logic. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 259–270.

Botinčan, M., Distefano, D., Dodds, M., Griore, R., Naudžiūnienė, and Parkinson, M. 2011. coreStar: The core of jStar. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs*. 65–77.

Burnim, J. and Sen, K. 2010. Asserting and checking determinism for multithreaded programs. *Comm. ACM 53*, 97–105.

Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM 58*, 6.

Calcagno, C., Distefano, D., and Vafeiadis, V. 2009. Bi-abductive resource invariant synthesis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 259–274.

Cook, B., Haase, C., Ouaknine, J., Parkinson, M. J., and Worrell, J. 2011. Tractable reasoning in a fragment of separation logic. In *Proceedings of the International Conference on Concurrency Theory*. Springer, 235–249.

Cook, B., Magill, S., Raza, M., Simsa, J., and Singh, S. 2010. Making fast hardware with separation logic. http://cs.cmu.edu/~smagill/papers/fast-hardware.pdf.

Creignou, N. and Zanuttini, B. 2006. A complete classification of the complexity of propositional abduction. *SIAM J. Comput. 36*, 1, 207–229.

Deshmukh, J. V., Ramalingam, G., Ranganath, V. P., and Vaswani, K. 2010. Logical concurrency control from sequential proofs. In *Proceedings of the European Symposium on Programming*. Springer, 226–245.

Distefano, D. and Filipović, I. 2010. Memory leaks detection in java by bi-abductive inference. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*. Springer, 278–292.

Distefano, D. and Parkinson, M. J. 2008. jStar: Towards practical verification for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 213–226.

Dodds, M., Jagannathan, S., and Parkinson, M. J. 2011. Modular reasoning for deterministic parallelism. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, 259–270.

Eiter, T. and Gottlob, G. 1995. The complexity of logic-based abduction. *J. ACM 42*, 1, 3–42.

Ghiya, R., Hendren, L. J., and Zhu, Y. 1998. Detecting parallelism in c programs with recursive data structures. In *Proceedings of the International Conference on Compiler Construction*. Springer, 159–173.

Golan-Gueta, G., Bronson, N. G., Aiken, A., Ramalingam, G., Sagiv, M., and Yahav, E. 2011. Automatic fine-grain locking using shape properties. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 225–242.

Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., and Sagiv, M. 2007. Local reasoning for storable locks and threads. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 19–37.

Gupta, R., Pande, S., Psarris, K., and Sarkar, V. 1999. Compilation techniques for parallel systems. *Parallel Comput. 25*, 13–14, 1741–1783.

Haack, C., Huisman, M., and Hurlin, C. 2008. Reasoning about Java's reentrant locks. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 171–187.

Hendren, L. J. and Nicolau, A. 1990. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst. 1*, 1, 35–47.

Hoare, C. A. R. and O'Hearn, P. W. 2008. Separation logic semantics for communicating processes. *Electron. Notes Theor. Comput. Sci. 212*, 3–25.

Hobor, A., Appel, A. W., and Zappa Nardelli, F. 2008. Oracle semantics for concurrent separation logic. In *Proceedings of the European Symposium on Programming*. Springer.

Horwitz, S., Pfeiffer, P., and Reps, T. W. 1989. Dependence analysis for pointer variables. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 28–40.

Hurlin, C. 2009. Automatic parallelization and optimization of programs by proof rewriting. In *Proceedings of the International Static Analysis Symposium*. Springer, 52–68.

Jacobs, B. and Piessens, F. 2009. Modular full functional specification and verification of lock-free data structures. Tech. rep. CW 551, Department of Computer Science, Katholieke Universiteit Leuven.

Leino, K. R. M., Müller, P., and Smans, J. 2010. Deadlock-free channels and locks. In *Proceedings of the European Symposium on Programming*. Springer, 407–426.

Liu, T., Curtsinger, C., and Berger, E. D. 2011. Dthreads: Efficient deterministic multithreading. In *Proceedings of the ACM Symposium on Operating Systems Principles*. ACM, 327–336.

Navabi, A., Zhang, X., and Jagannathan, S. 2008. Quasi-static scheduling for safe futures. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 23–32.

O'Hearn, P. W. 2007. Resources, concurrency and local reasoning. *Theor. Comput. Sci. 375*, 271–307.

Ottoni, G., Rangan, R., Stoler, A., and August, D. I. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 105–118.

Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. 2011. The tao of parallelism in algorithms. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 12–25.

Raychev, V., Vechev, M., and Yahav, E. 2013. Automatic synthesis of deterministic concurrency. In *Proceedings of the International Static Analysis Symposium*. Springer.

Raza, M., Calcagno, C., and Gardner, P. 2009. Automatic parallelization with separation logic. In *Proceedings of the European Symposium on Programming*. Springer, 348–362.

Reynolds, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

SV-Comp. 2013. http://sv-comp.sosy-lab.org/2013/.

Tang, P., Tang, P., Zigman, J. N., and Zigman, J. N. 1994. Reducing data communication overhead for DOACROSS loop nests. In *Proceedings of the International Conference on Supercomputing*. ACM, 44–53.

Villard, J., Lozes, É., and Calcagno, C. 2010. Tracking heaps that hop with Heap-Hop. In *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 275–279.

Welc, A., Jagannathan, S., and Hosking, A. 2005. Safe futures for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 439–435.

Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P. W. 2008. Scalable shape analysis for systems code. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 385–398.

Received February 2012; revised January 2013; accepted March 2013