Caper

Automatic Verification for Fine-grained Concurrency

Thomas Dinsdale-Young[1], Pedro da Rocha Pinto[2], Kristoffer Just Andersen[1], and Lars Birkedal[1]

[1] Aarhus University, Aarhus, Denmark ({tyoung,kja,birkedal}@cs.au.dk)
[2] Imperial College London, London, UK (pmd09@doc.ic.ac.uk)

Abstract. Recent program logics based on separation logic emphasise a modular approach to proving functional correctness for fine-grained concurrent programs. However, these logics have no automation support. In this paper, we present Caper, a prototype tool for automated reasoning in such a logic. Caper is based on symbolic execution, integrating reasoning about interference on shared data and about ghost resources that are used to mediate this interference. This enables Caper to verify the functional correctness of fine-grained concurrent algorithms.

1 Introduction

In recent years, much progress has been made in developing program logics for verifying the functional correctness of concurrent programs [10,32,29,7,19], with emphasis on fine-grained concurrent data structures. Reasoning about such programs is challenging since data is concurrently accessed by multiple threads: the reasoning must correctly account for interference between threads, which can often be subtle. Recent program logics address this challenge by using *resources* that are associated with some form of *protocol* for accessing shared data.

The concept of heap-as-resource was a fundamental innovation of separation logic [26]. It is possible to specify and verify a piece of code in terms of the resources that it uses. Further resources, which are preserved by the code, can be added by framing, provided that they are disjoint. Concurrent separation logic (CSL) [23] uses the observation that threads operating on disjoint resources do not interfere. This is embodied in the disjoint concurrency proof rule:

$$\frac{\{p_1\}\ c_1\ \{q_1\} \qquad \{p_2\}\ c_2\ \{q_2\}}{\{p_1 * p_2\}\ c_1 \| c_2\ \{q_1 * q_2\}}\ .$$

The *separating conjunction* connective '$*$' in the assertion $p_1 * p_2$ asserts that both p_1 and p_2 hold but for disjoint portions of the heap. In separation logic, the primitive resources are heap cells, represented $x \mapsto y$, meaning the heap at address x holds value y. A thread that owns a heap cell has an exclusive right to read, modify or dispose of it. The separating conjunction $x \mapsto y * y \mapsto z$ enforces disjointness: it requires x and y to be different addresses in the heap.

In fine-grained concurrent algorithms, threads share access to data, as opposed to using disjoint data, so a more flexible concept of resources is required. One approach to this is the use of *shared regions* [10]. A shared region encapsulates some underlying (concrete) resources, which may be accessed by multiple threads when they perform atomic operations. The region enforces a protocol that determines how threads can mutate the resources that it encapsulates. The region is associated with abstract resources called *guards* that determine the role that a thread can play in the protocol. Importantly, these resources determine what knowledge a thread can have about the region that is *stable* — i.e., continues to hold under the actions of other threads.

For example, consider a region that encapsulates a heap cell at address x. Associated with this region are two guards Inc and Dec. The protocol states that a thread with the Inc guard can increase the value stored at x, and a thread with the Dec guard can decrease the value stored at x. A thread holding the Inc guard can know that the value at x is *at most* the last value it observed; without the Dec guard, a thread cannot know that the value will not be decreased by another thread. Conversely, a thread holding the Dec guard can know a lower bound on the value at x. A thread that holds both guards can change the value arbitrarily and know it precisely, much as if it had the resource $x \mapsto y$ itself.

In this paper we present Caper, the first tool for automatic verification of fine-grained concurrent programs using separation logic. To verify a program, the user specifies the types of shared regions, defining their guards and protocols, and provides specifications for functions (and loop invariants) that use these regions. Caper uses a *region-aware* symbolic execution (§3.3) to verify the code, in the tradition of SmallFoot [1]. The key novelties of Caper's approach are:

- the use of *guard algebras* (§3.1) as a mechanism for representing and reasoning automatically about abstract resources, while supporting a range of concurrency verification patterns;
- techniques for automatically reasoning about interference on shared regions (§3.2), in particular, accounting for transitivity; and
- heuristics for non-deterministic proof search (§4), including the novel use of abduction to infer abstract updates to shared regions and guards.

We introduce our approach by considering a number of examples in §2. We emphasise that these examples are complete and self-contained — Caper can verify them without additional input. In §5 we evaluate Caper, reporting results for a range of examples. We discuss related work in §6 before concluding with remarks on future directions in §7.

The Caper tool is implemented in Haskell, and uses Z3 [8] and (optionally) E [27] to discharge proof obligations. The source code and examples are available [11], as is a soundness proof of the separation logic underlying Caper [12].

2 Motivating Examples

We begin by considering a series of examples that illustrate the programs and specifications that Caper is designed to prove. In each case, we discuss how examples/recursive/SpinLock.t

```
1  region SLock(r,x) {
2    guards %LOCK * UNLOCK;
3    interpretation {
4      0 : x ↦ 0 &*& r@UNLOCK;
5      1 : x ↦ 1;
6    }
7    actions {
8      LOCK[_] : 0 ⇝ 1;
9      UNLOCK : 1 ⇝ 0;
10   }
11 }
12 function makeLock()
13   requires true;
14   ensures SLock(r,ret,0) &*& r@LOCK[1p]; {
15     v := alloc(1);
16     [v] := 0;
17     return v;
18 }
19 function acquire(x)
20   requires SLock(r,x,_) &*& r@LOCK[p];
21   ensures SLock(r,x,1) &*& r@(LOCK[p] * UNLOCK); {
22     b := CAS(x, 0, 1);
23     if (b = 0) {
24       acquire(x);
25     }
26 }
27 function release(x)
28   requires SLock(r,x,1) &*& r@UNLOCK;
29   ensures SLock(r,x,_); {
30     [x] := 0;
31 }
```

Fig. 1. Caper listing for a spin lock implementation.

Caper handles the example and why. In later sections, we will describe the rules and algorithms that underlie Caper in more detail. For each example, we give the *complete* source file, which Caper verifies with no further annotation.

2.1 Spin Lock

Fig. 1 shows a typical annotated source file for Caper, which implements a simple fine-grained concurrent data structure: a spin lock. Note that `&*&` is Caper syntax for $*$ — separating conjunction.

Lines 1–11 define a *region type*, `SLock`, for spin locks. There are two kinds of assertions associated with regions, and their shape is dictated by region type definitions. `SLock(r,x,s)` is the assertion representing knowledge of a region `r` of type `SLock` with parameter `x` in abstract state `s`. The second are guard assertions of the form `r@(G)`, meaning we hold the guard `G` for region `r`.

Line 2 is a *guard algebra* declaration, indicating the guards associated with a given region type, and how they compose. There are two kinds of guard associated with `SLock` regions: `LOCK` guards, which are used to acquire the lock, and may be subdivided to allow multiple threads to compete for the lock (indicated by `%` in the `guards` declaration); and `UNLOCK` guards, which are used to release the lock, and are exclusive — only a thread holding the lock owns the `UNLOCK` guard.

Lines 3–6 declare a *region interpretation*: the resources held by the region when in each abstract state. `SLock` regions have two states: 0 represents that the lock is *available*, which is indicated concretely by the heap cell $\mathtt{x} \mapsto 0$; 1 represents that the lock has been *acquired*, which is indicated concretely by the heap cell $\mathtt{x} \mapsto 1$. In the *available* state the `UNLOCK` guard belongs to the region — a thread that transitions to the acquired state obtains this guard.

Finally, lines 7–10 declare the *actions* — the protocol governing the shared region. This embodies both the updates allowed for a given thread and the interference a thread must anticipate from the environment. A thread can transition an `SLock` region from abstract state 0 (available) to 1 (acquired) if it holds the `LOCK[p]` guard for any `p`. Similarly, a thread can transition an `SLock` region from acquired to available if it holds the `UNLOCK` guard.

The `makeLock` function allocates a new spin lock. It has the precondition `true` since it does not require any resources to operate. In the postcondition, the function returns an `SLock` region with full permission to its `LOCK` guard (the latter is expressed by the `r@(LOCK[1p])` assertion). The logical variable `r` holds the identifier for the region, which is used to relate assertions that refer to the same region; it is implicitly existentially quantified as it occurs in the postcondition but neither in the precondition nor as a parameter to the function. The logical variable `ret` binds the return value, which is the address of the lock. When Caper symbolically executes the function body, at the `return` it has the resource $\mathtt{v} \mapsto 0$ but requires $\mathtt{SLock}(r, \mathtt{v}, 0)$. This missing resource is abduced, which means that Caper backtracks searching for a place where it could have obtained the missing resource. This causes Caper to try to construct the region before executing the `return` statement. Constructing the region consists of creating a fresh region identifier and adding the full guards for the region to the symbolic state (in this case `LOCK[1p] * UNLOCK`); the resources belonging to the region according to the interpretation are consumed (removed from the symbolic state). This is successful for the interpretation 0, leaving behind the guard `LOCK[1p]` for the new region. Caper can then successfully symbolically execute the `return` statement, since it has the resources required to satisfy the postcondition.

The `acquire` function attempts to acquire the lock. The precondition asserts that the spin lock is in an unknown state and that we have permission to acquire the lock in the form of the $\mathtt{LOCK[}p\mathtt{]}$ guard. The postcondition asserts that the lock is in the acquired state (indicated by the 1 in the `SLock` predicate) and that we retain the $\mathtt{LOCK[}p\mathtt{]}$ guard but have also acquired the `UNLOCK` guard.

The lock is acquired by performing an atomic compare-and-set (CAS) operation, which attempts to set the value stored at address `x` from 0 to 1. In symbolically executing the CAS, Caper determines that it needs to open the region because it does not have $\mathtt{x} \mapsto -$. In opening the region, Caper branches on the interpretation of the region; it must show that both cases are correct. The CAS itself also introduces branches depending on whether it failed; these are quickly pruned as the CAS cannot fail if the region is in state 0, nor succeed if it is in state 1. Immediately after the atomic CAS, Caper must close the region. It does so by non-deterministically choosing among the interpretations.

If the initial state was 1, the CAS fails and Caper closes with state 1. Since the state is unchanged, this 'update' is permitted. After the atomic operation, Caper must stabilise the region; since the thread does not own the `UNLOCK` guard, another thread could transition the region to state 0. Consequently, after the CAS, Caper does not know which state the region is in. Since the CAS fails, the `if` condition succeeds. Caper then makes the recursive call to `acquire` using the specification, which allows it to obtain the postcondition in the end.

If the initial state was 0, the CAS succeeds and Caper closes with state 1. In doing so, the `UNLOCK` guard is acquired, since it is part of the interpretation of state 0, but not of state 1. Caper must then check that the update from state 0 to 1 is permitted by the available guards, `LOCK[`p`] * UNLOCK`, which it is. After the CAS, the thread owns the `UNLOCK` guard so no other thread can change the state of the region, and so it is stable in state 1. The result of a successful CAS is 1, so Caper does not symbolically execute the body of the `if` in this case, and proceeds to check the postcondition, which is successful.

The verification of the `release` function proceeds along similar lines.

2.2 Ticket Lock

Fig. 2 shows a Caper listing for a ticket lock. A ticket lock comprises two counters: the first records the next available ticket and the second records the ticket which currently holds the lock. (Note that the lock is "available" when the two counters are equal — i.e. the ticket holding the lock is the next available ticket.) To acquire the lock, a thread obtains a ticket by incrementing the first counter and waiting until the second counter reaches the value of the ticket it obtained. To release the lock, a thread simply increments the second counter.

In Caper, a ticket lock is captured by a `TLock` region, defined in lines 1–9 of Fig. 2. In contrast to the `SLock` region, a `TLock` region has an infinite number of states: there is an abstract state for each integer `n`. The abstract state `n` of a `TLock` region represents the ticket that currently holds the lock. The guards associated with a `TLock` region represent the tickets: there is a unique guard `TICKET(n)` for each integer `n`. (This is indicated by the # in the `guards` declaration.) The region interpretation of state `n` ensures that:

- the first counter (`x`) is the next available ticket number, `m`, which is at least `n`;
- the second counter (`x+1`) is `n`, the lock-holding ticket number;
- all `TICKET` resources from `m` up belong to the region. (A set of indexed resources is expressed with a set-builder-style notation, as in `TICKET{k|k`$\geq$`m}`.)

Note that `m` is implicitly existentially quantified in the interpretation.

A thread may acquire a ticket by incrementing the next-available-ticket counter and removing the corresponding `TICKET` guard from the region. Doing so does not affect the abstract state of the region, and can, therefore, happen at any time (no guards are required to do so). In order to increment the lock-holding-

```
examples/iterative/TickLock.t
region TLock(r,x) {
  guards #TICKET;
  interpretation {
    n : x ↦ m &*& (x + 1) ↦ n &*& r@TICKET{ k | k ≥ m } &*& m ≥ n;
  }
  actions {
    n < m | TICKET{ k | n ≤ k, k < m } : n ⇝ m;
  }
}
function acquire(x)
  requires TLock(r,x,_);
  ensures TLock(r,x,n) &*& r@TICKET(n); {
    do {
        t := [x + 0];
        b := CAS(x + 0, t, t + 1);
    }
      invariant TLock(r,x,ni) &*& (b=0 ? true : r@TICKET(t) &*& t ≥ ni);
    while (b = 0);
    do {
        v := [x + 1];
    }
      invariant TLock(r,x,ni) &*& r@TICKET(t) &*& t ≥ ni &*& ni ≥ v;
    while (v < t);
}
function release(x)
  requires TLock(r,x,n) &*& r@TICKET(n);
  ensures TLock(r,x,_); {
    v := [x + 1];
    [x + 1] := v + 1;
}
```

Fig. 2. Caper listing for a ticket lock implementation.

ticket counter, a thread must hold the `TICKET(n)` resource for the current value of the counter, `n`. We might, therefore, expect the `actions` declaration to be:

```
  actions {
    TICKET(n) : n ⇝ n + 1;
  }
```

This action declaration is, however, problematic for automation. Between symbolically executing atomic actions, Caper widens the set of possible abstract states for each region according to the *rely* relation for that region type. Suppose a `TLock` region is initially in state 0. If the thread does not hold the `TICKET(0)` guard, Caper must add the state 1 to the possible state set. If the thread does not hold `TICKET(1)`, Caper must add the state 2, and so on. In general, we cannot expect this widening process to terminate, so we must consider a *transitively-closed* rely relation. Caper cannot, in general, compute the transitive closure, but it *is* possible to *check* that a given `actions` declaration is transitively closed. We address this in §3.2. The proposed action declaration is, however, not transitive, since transitions from 0 to 1 and from 1 to 2 are possible, but the transitive transition from 0 to 2 is not possible in one step.

Instead, we use the actions declaration in Fig. 2, which *is* transitively closed. It remedies the problem with the simple version by generalising from a single increment to allow multiple increments. This is achieved by placing a condition on the transition that $\texttt{n}\rightsquigarrow\texttt{m}$ is only permitted when n<m, enforcing that the counter can only increase, as indicated before the vertical bar in the actions declaration. The guard $\texttt{TICKET}\{\texttt{k}|\texttt{n}\leq\texttt{k},\texttt{k}<\texttt{m}\}$ denotes the set of all guards TICKET(k) for k between n and m-1 inclusive. This ensures that a thread can increment the counter past k only when it holds the TICKET(k) guard. For example, a thread holding TICKET(n) can transition the TLock region from abstract state n to n+1.

The precondition of acquire asserts that the ticket lock exists in some arbitrary abstract state. The postcondition ensures that the lock is in some state n and that the guard TICKET(n) has been acquired. The function contains two loops and Caper requires that we provide an invariant for each. The first loop, lines 13–18, increments the next available ticket. The invariant states that the region is in some state ni and that once the CAS succeeds ($\texttt{b} = 1$) we have the TICKET(t) guard and t is at least ni; the conditional is expressed using the C-like _?_:_ notation. Similarly to the acquire operation for the spin lock, Caper opens the region when symbolically executing the CAS operation. Since there is only one clause in the TLock region interpretation, Caper considers one generic case, rather than branching as in the spin lock. Immediately after symbolically executing the CAS operation, Caper needs to close the region. If the CAS succeeds Caper knows that the next available ticket m is t+1. Hence the guard TICKET(t) is not included in the set of guards $\texttt{TICKET}\{\texttt{k}|\texttt{k}\geq\texttt{m}\}$ needed for closing the region, so Caper can transfer the guard TICKET(t) out of the TLock region. The next loop, lines 19–23, spins until the acquired ticket becomes the lock-holding ticket. Caper proceeds similarly to the first loop. After the loop, the invariant and failed loop test are sufficient to establish the postcondition.

The precondition of the release function expresses that the lock-holding ticket of the region is n and that we hold that ticket. Because we hold the guard TICKET(n), we can make a transition from abstract state n to abstract state n+1. No other thread can make a transition, since to transition from n to m one needs to hold all the guards from n to m-1. Therefore, there is no interference from other threads on the second counter and we can update it without using a CAS loop. After the read on line 28, Caper knows that v holds value n by opening the region. To execute the write on line 29, Caper again opens the region in state n. Caper closes the region in a new state $\texttt{n}_1$, which must be the value of the x+1 counter, i.e. $\texttt{n}_1 = \texttt{n} + 1$. The value of m is unchanged, but Caper must establish that $\texttt{m} \geq \texttt{n}_1 = \texttt{n} + 1$, which follows from the fact that $\texttt{m} \geq \texttt{n}$ and that TICKET(n) (from the thread) is disjoint from $\texttt{TICKET}\{\texttt{k}|\texttt{k}\geq\texttt{m}\}$ (from the region). Caper must also establish that the transition $\texttt{n}\rightsquigarrow\texttt{n+1}$ is permitted by the actions for the available guards, which it is.

Client. Fig. 3 shows an implementation of a simple client using the ticket lock. Here, the Client(r,x,s,z) region uses a ticket lock region TLock(s,z) to maintain the lock invariant that two cells, x and x+1, have the same value. The examples/iterative/TickLockClient.t

```
1  region Client(r,x,s,z) {
2    guards 0;
3    interpretation {
4      0 : TLock(s,z,k) &*& (x ↦ a &*& (x+1) ↦ a ∨ s@TICKET(k));
5    }
6    actions {}
7  }
8  function set(x,z,w)
9    requires Client(r,x,s,z,0) &*& TLock(s,z,_);
10   ensures Client(r,x,s,z,0) &*& TLock(s,z,_); {
11     acquire(z);
12     [x] := w;
13     [x + 1] := w;
14     release(z);
15 }
```

Fig. 3. A Caper listing of a client of the ticket lock.

disjunction with the guard s@TICKET(k) makes it possible to temporarily break the invariant. The function set(x,z,w) sets the value of the two shared memory cells to w. Lines 12–13 are a critical section protected by the ticket lock. Note that the invariant is temporarily broken between the two writes.

In symbolically executing the call to acquire in line 11, Caper uses the postcondition of acquire to obtain s@TICKET(k) and TLock(s,z,k) for some k. When Caper symbolically executes line 12, it opens the Client region and must consider each of the disjuncts of the interpretation. It finds that the right-hand disjunct is not possible as we already hold the TICKET guard for the current abstract state k of the lock. Hence it obtains the points-to assertions for x and x+1, and can perform the write to x. Since the values stored at x and x+1 are now different, Caper can only close the region by transferring s@TICKET(k) to the region. When Caper symbolically executes line 13, it again opens the Client region. This time it finds that the region holds s@TICKET(k) since we have the points-to predicates. After the assignment, the values stored at x and x+1 are the same, and Caper can close the region while leaving us with the s@TICKET(k) resource which Caper then uses to satisfy the precondition of release.

2.3 Stack-based Bag

Fig. 4 shows a Caper implementation of a concurrent bag based on Treiber's stack [31]. The stack is lock-free, and uses CAS operations to manipulate the head of a linked-list structure. To push a new item, a thread constructs a new head node and atomically updates the head pointer of the bag. When popping an item, a thread anticipates what the head node is before atomically updating the head pointer. In both cases, the function of the atomic compare-and-swap operation is to ensure that no other thread has manipulated the bag between operations. Unlike the preceding examples, the heap is fundamental to the implementation.

The specification is parametrised by an *abstract predicate* [24,2] bagInvariant (line 1). The idea is that adding an item v to the bag requires transferring examples/recursive/BagStack.t

```
predicate bagInvariant(v);
region Bag(r,x) {
  guards 0;
  interpretation {
    0 : x ↦ y &*& BagList(s,y,_,_,0) &*& s@OWN;
  }
  actions {}
}
region BagList(s,y,v,z) {
  guards OWN;
  interpretation {
    0 : y = 0 ? true : y ↦ v &*& y+1 ↦ z
        &*& BagList(t,z,_,_,0) &*& t@OWN &*& bagInvariant(v);
    1 : s@OWN &*& y ↦ v &*& y+1 ↦ z &*& BagList(t,z,_,_,_);
  }
  actions {
    OWN : 0 ⇝ 1;
  }
}
function push(x,v)
  requires Bag(r,x,0) &*& bagInvariant(v);
  ensures Bag(r,x,0); {
    y := alloc(2); [y] := v;
    innerPush(x,y);
}
function innerPush(x,y)
  requires Bag(r,x,0) &*& y ↦ v &*& y+1 ↦ _ &*& bagInvariant(v);
  ensures Bag(r,x,0); {
    t := [x];
    [y + 1] := t;
    cr := CAS(x,t,y);
    if (cr = 0) {
      innerPush(x, y);
    }
}
function pop(x)
  requires Bag(r,x,0);
  ensures ret = 0 ? Bag(r,x,0) : Bag(r,x,0) &*& bagInvariant(ret); {
    t := [x];
    if (t = 0) { return 0; }
    t2 := [t + 1];
    cr := popCAS(x,t,t2);
    if (cr = 0) { ret := pop(x); return ret; }
    ret := [t];
    return ret;
}
function popCAS(x,t,t2)
  requires Bag(r,x,0) &*& BagList(rt,t,v,t2,_)
          &*& BagList(rt2,t2,_,_,_) &*& t != 0;
  ensures ret = 0 ∨ bagInvariant(v); {
    cr := CAS(x,t,t2); return cr;
}
```

Fig. 4. Caper listing for a concurrent bag implementation.

ownership of the predicate `bagInvariant`(v) to the bag, which is returned when the item is removed. Clients can decide how to instantiate `bagInvariant`.

In Caper, the head pointer and the linked-list nodes are encapsulated by separate regions: `Bag` and `BagList`, respectively. Note that there is an apparent hierarchy between `Bag` and `BagList` regions: a `Bag` refers to a `BagList`, which may, in turn, refer to another `BagList` and so on. In this way, we can use regions to model inductive data structures such as linked lists. While regions can fulfil a similar role to inductive predicates, they are semantically distinct. Regions are shared globally, and so may refer to each other in arbitrary, even cyclical ways. Although it appears as though the regions are nested, semantically all regions exist at the same level. In this example, we achieve a hierarchy through ownership of guards: the top-level `Bag` holds the `OWN` guard for the first `BagList`, which holds the `OWN` guard for the second, and so on.

A `Bag` region is simple, in that it has no guards and is always in one abstract state. It simply permits sharing of the resources it holds. The interpretation of abstract state 0 of a region `Bag(r,x)` holds a pointer to the first, possibly null, linked-list node at `x` in addition to its `OWN` guard.

The `BagList(s,y,v,z)` region type represents a list node (or a null terminator) with payload value `v` at address `y` and a pointer to the successor `z` at `y+1`. A `BagList` region is in one of two states, depending on whether it belongs to the bag or not. Abstract state 0 means that the region belongs to the bag, in which case it can represent either a null-pointer terminating the list or a list node with a value and successor, represented by another `BagList` region. The region also holds the `OWN` guard to the successor and the `bagInvariant` predicate for its value. The abstract state 1 represents a list node that has been popped. The interpretation therefore includes the region's own `OWN` guard and knowledge of the successor, but *not* the successor's `OWN` guard or the `bagInvariant` predicate.

Since the bag can be used concurrently, we do not specify exactly which elements it contains at a given time. Instead, our specification of `push` and `pop` focuses on ownership transfer of elements pushed to and popped from the bag.

The precondition of `push` asserts only knowledge of the bag and ownership of the `bagInvariant` for the value `v` to be put in the bag. In the postcondition, the `bagInvariant` resource is absent, as it is transferred to the bag. The `push` function allocates a new list node and then delegates to a CAS loop in `innerPush`.

The specification of `innerPush` is similar to that of `push`, but the precondition requires the list node that is to be pushed. Note that the successor of the node, `y+1`, is initially undetermined. In lines 29–30, `innerPush` loads the current head of the list into the tail pointer of the new node via the variable `t`. To do so, Caper opens the `Bag` region, getting access to `x`, and closes it again. The observed value is then written as the successor of the new node, at address `y+1`, without opening any regions. To account for the head of the stack having changed since it was read, a CAS is used to update the head pointer. To symbolically execute the CAS in line 31, Caper again opens the `Bag` region. If successful, it must close and restore the `Bag` region. This means a new `BagList` region in state 0 must be created for the new head. Upon creation, Caper creates the `OWN` guard for the new region, which is given to the Bag region, closing it in state 0. If the CAS does not succeed, it means another thread updated the stack between lines 29 and 31, and innerPush recursively tries again.

The specification of the pop function states that, from a Bag, pop produces either null (0), in case the bag is empty, or a value satisfying the bagInvariant predicate. The idea is that the concrete value comes from the underlying linked list, and the corresponding bagInvariant(v) predicate is removed from a BagList region. The pop function attempts to CAS the head pointer of the bag to the successor of the first link, effectively removing the first element from the bag. It first reads the head pointer into t, which requires opening the Bag region. At this point, we obtain the BagList region, which can be freely duplicated, although the OWN guard remains in the Bag region. After the Bag region is closed again, we must stabilise the BagList region to account for the fact that another thread could remove the head node from the stack. That is, its abstract state could now be 0 or 1. If the head pointer was 0, then the bag was empty and so 0 is returned. Otherwise, t points to a node, and line 41 reads its successor pointer into t2. This involves opening the BagList region previously obtained. It is not necessary to open the Bag region again at this stage. The call to popCAS at line 42 attempts to update the head node to the successor of the head node. We give this CAS operation a specification (lines 48–50) to assist Caper. To symbolically execute that CAS, Caper opens the Bag region containing x and the OWN guard for the BagList region for the head of the stack. The BagList region for the previously seen head (with region identifier rt) and the BagList region for the current head (if it is different) are also opened. Caper determines that the CAS can only succeed if these two regions are the same. In this case, the bagInvariant is transferred to the thread, the OWN guard for the successor is transferred to the Bag region, and the OWN guard for the head is transferred to its own BagList region. In this process, the state of the BagList region for the old head is updated from 0 to 1. This is allowed since we have access to its OWN guard. If the CAS fails, nothing is changed and the pop is retried. On success, the return value is read from the BagList region that is now in state 1. This value corresponds to the previously-obtained bagInvariant predicate.

3 Proof System

Caper's proof system is based on the logic of CAP [10], using improvements from iCAP [29] and TaDA [7]. The logic is a separation logic with shared regions.

Each shared region has a unique *region identifier*. A region has an associated *region type*, which determines the resources and protocol associated with the region. Region types $\mathbf{T}(r, \bar{x})$ are parametrised, with the first parameter (r) always being the region identifier. A region also has an *abstract state*. In Caper's logic, *region assertions* describe the type and state of a region. The region assertion $\mathbf{T}(r, \bar{x}, y)$ asserts the existence of a region with identifier r and type $\mathbf{T}(r, \bar{x})$ in abstract state y. Region assertions are freely duplicable; i.e., they satisfy the equivalence: $\mathbf{T}(r, \bar{x}, y) \iff \mathbf{T}(r, \bar{x}, y) * \mathbf{T}(r, \bar{x}, y)$. Moreover, region assertions

$$\frac{(P(\bar{z})|G : e_1(\bar{z}) \rightsquigarrow e_2(\bar{z})) \in \mathrm{Actions}(\mathbf{T}(r, \bar{x})) \qquad P(\bar{w})}{(e_1(\bar{w}), e_2(\bar{w})) \in \mathrm{Guar}(\mathbf{T}(r, \bar{x}), G)} \qquad \frac{}{(x, x) \in \mathrm{Guar}(\mathbf{T}(r, \bar{x}), G)}$$

$$\frac{(x, y), (y, z) \in \mathrm{Guar}(\mathbf{T}(r, \bar{x}), G)}{(x, z) \in \mathrm{Guar}(\mathbf{T}(r, \bar{x}), G)}\,(\dagger) \qquad \frac{(x, y) \in \mathrm{Guar}(\mathbf{T}(r, \bar{x}), G)}{(x, y) \in \mathrm{Guar}(\mathbf{T}(r, \bar{x}), G * G')}$$

$$\frac{(P(\bar{z})|G : e_1(\bar{z}) \rightsquigarrow e_2(\bar{z})) \in \mathrm{Actions}(\mathbf{T}(r, \bar{x})) \qquad P(\bar{w}) \qquad G * G' \text{ defined}}{(e_1(\bar{w}), e_2(\bar{w})) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G')}$$

$$\frac{(x, y), (y, z) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)}{(x, z) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)}\,(\dagger) \qquad \frac{(x, y) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G * G')}{(x, y) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)}$$

$$\frac{}{(x, x) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)}$$

Fig. 5. Rules defining the Guar and Rely relations.

with the same region identifier must agree on the region type and abstract state: $\mathbf{T}(r, \bar{x}, y) * \mathbf{T}'(r, \bar{x}', y') \implies (\mathbf{T}, \bar{x}, y) = (\mathbf{T}', \bar{x}', y')$.

Shared regions are also associated with (ghost) resources called *guards*. Which guards can be associated with a region, as well as their significance and behaviour is determined by the type of the region. Guards are interpreted as elements of a partial commutative monoid (PCM), referred to as a *guard algebra*. That is, they have a partial composition operator that is associative, commutative and has a unit. This is sufficient for them to behave as separation logic resources (see e.g. [9]). In Caper's logic, *guard assertions* assert ownership of guard resources. The guard assertion $r@(G_1 * \ldots * G_n)$ asserts ownership of the guards $G_1, \ldots, G_n$ associated with region identifier r. Guard assertions are not in general duplicable, but they distribute with respect to $*$: $r@(G_1 * G_2) \iff r@G_1 * r@G_2$.

The region type determines how a shared region is used. A region type definition determines the following properties of regions of that type: the guard algebra associated with the region; the abstract states of the region and their concrete *interpretation*; and the *actions* that can be used to update the state of the region, and which guards are required in order to perform each action. Region type definitions determine two derived relations, Rely and Guar (for *guarantee*), which are defined in Fig. 5, based on the actions for the region types. The relation $\mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)$ consists of all state transitions that a thread's environment may make to a region of type $\mathbf{T}(r, \bar{x})$, if the thread owns guard G. The relation $\mathrm{Guar}(\mathbf{T}(r, \bar{x}), G)$ consists of all state transitions that a thread itself may make to a region of type $\mathbf{T}(r, \bar{x})$, if it owns guard G.

3.1 Guards

The underlying logic of Caper permits the guard algebra for a region to be an arbitrary PCM. However, in order to reason automatically about guards, Caper must be able to effectively compute solutions to certain problems within the PCM. To this end, Caper provides a number of constructors for guard algebras for which these problems are soluble. These constructors are inspired by common patterns in concurrency verification, and are useful for many examples. The three automation problems are as follows.

Frame Inference. Given guard assertions `A` and `B`, find a `C` (if it exists) such that `A` $\vdash$ `B` $*$ `C`. This problem has two applications: 1. Computing the Guar relation requires determining if the guard currently available to the thread (`A`) entails the guard required to perform some action (`B`); 2. At call sites, returns and when closing regions, symbolic execution consumes assertions (i.e. checks that the assertion holds and removes the corresponding resources from the symbolic state). Frame inference (for guards) does this for guard assertions.

Composition and Compatibility. Given guard assertions `A` and `B`, determine their composition `A` $*$ `B` and the condition for it being defined. Whether it is defined will be a pure assertion on the free variables of `A` and `B`. This also has two applications: 1. Computing the Rely relation requires determining when the guard currently owned by the thread (`A`) is compatible with the guard required to perform some action (`B`); 2. At entry points, after calls and when opening regions, symbolic execution produces assertions (i.e. adds resources and assumptions corresponding to the assertion to the symbolic state). Composition does this for guards.

Least Upper-bounds. Given guard assertions `A` and `B`, compute `C` such that `C` $\vdash$ `A`, `C` $\vdash$ `B` and for any `D` with `D` $\vdash$ `A` and `D` $\vdash$ `B`, `D` $\vdash$ `C`. This is used to compose two actions, which will be guarded by the least upper-bound of the two guards.

Supported Guard Algebras. We present the guard algebras that are supported by CAPER. Each guard algebra has a maximal element, the *full guard*, which is generated for a region when it is initially created.

Trivial guard algebra. The trivial guard algebra, `0` in CAPER syntax, consists of one element which is the unit. This algebra is used when a region has no roles associated with it: it can be used in the same way by all threads at all times.

All-or-nothing guard algebra. An all-or-nothing guard algebra consists of a single element distinct from the unit, which is the full guard. In CAPER syntax, this algebra is represented by the name chosen for the point; for instance, `GUARD` would be the all-or-nothing algebra with point `GUARD`.

Permissions guard algebra. A permissions guard algebra `%GUARD` has the full resource `GUARD[1p]` which can be subdivided into smaller permissions `GUARD[`π`]`. The typical model for permissions is as fractions in the interval $[0, 1]$. This allows for (non-zero) permissions to be split arbitrarily often. CAPER implements a different theory of permissions. This theory also allows arbitrary splitting, but also requires that `GUARD[`π`]` $*$ `GUARD[`π`]` is undefined for any non-zero π.

The theory can be encoded into the theory of *atomless Boolean algebras* — a Boolean algebra is said to be atomless if for all $a > \bot$ there exists a' with

$a > a' > \bot$. The encoding defines the PCM operator as $p_1 * p_2 = p_1 \vee p_2$ if $p_1 \wedge p_2 = \bot$ (and undefined otherwise). Conveniently, the first-order theory of atomless Boolean algebras is complete and therefore decidable (initially reported by Tarski [30], proved by Ershov [15], and see e.g. [6] for details).

Caper implements three different proof procedures for the theory of permissions. One uses the encoding with Boolean algebras and passes the problem to the first-order theorem prover E [27]. A second checks for the satisfiability of a first-order permissions formula directly. The third encodes the satisfiability problem with bit-vectors and passes it to the SMT solver Z3 [8]. Dockins et al. [13] previously proposed a tree-share model of permissions, which is also a model of this theory. Le et al. [20] have developed decision procedures for entailment checking based on this model, which could also be used by Caper.

Counting guard algebra. A counting guard algebra $\texttt{|GUARD|}$ consists of *counting guards* similar to the counting permissions of Bornat et al. [3]. For $n \geq 0$, $\texttt{GUARD|}n\texttt{|}$ expresses n counting guards. An *authority guard* tracks the number of counting guards that have been issued. For $n \geq 0$, $\texttt{GUARD|-1} - n\texttt{|}$ expresses the authority guard with n counting guards issued. The PCM operator is defined as:

$$\begin{aligned}\texttt{GUARD|}n\texttt{|} * \texttt{GUARD|}m\texttt{|} = \texttt{GUARD|}n+m\texttt{|} \quad & \text{if } (n \geq 0 \wedge m \geq 0) \vee \\ (n < 0 \wedge m \geq 0 \wedge n+m < 0) \vee {} & (n \geq 0 \wedge m < 0 \wedge n+m < 0).\end{aligned}$$

This ensures that the authority is unique (e.g. $\texttt{GUARD|-1|} * \texttt{GUARD|-2|}$ is undefined) and that owning $\texttt{GUARD|-1|}$, which is the full guard, guarantees that no other thread may have a counting guard.

Indexed guard algebra. An indexed guard algebra $\texttt{\#GUARD}$ consists of sets of individual guards $\texttt{GUARD(}n\texttt{)}$ where n ranges over integers. A set of such individual guards is expressed using a set-builder notation: $\texttt{GUARD}\{x \mid P\}$ describes the set of all guards $\texttt{GUARD}(x)$ for which P holds of x. The full guard is $\texttt{GUARD}\{x \mid \texttt{true}\}$. The notation $\texttt{GUARD}(n)$ is syntax for $\texttt{GUARD}\{x \mid x = n\}$. The PCM operator is defined as $\texttt{GUARD}S_1 * \texttt{GUARD}S_2 = \texttt{GUARD}(S_1 \cup S_2)$ if $S_1 \cap S_2 = \emptyset$.

The automation problems reduce to testing conditions concerning sets, specifically set inclusion. Sets in Caper are not first-class entities: they are always described by a logical predicate that is a (quantifier-free) arithmetic formula. For sets characterised in this way, set inclusion can be characterised as: $\{x \,|\, P\} \subseteq \{x \,|\, Q\} \iff \forall x.\, P \Rightarrow Q$. Advanced SMT solvers, such as Z3 [8], have support for first-order quantification, and Caper exploits this facility to handle the verification conditions concerning set inclusion.

Product guard algebra construction. Given guard algebras M and N, the product construction $M \mathbin{\texttt{*}} N$ consists of pairs $(m, n) \in M \times N$, where the PCM operation is defined pointwise: $(m_1, n_1) * (m_2, n_2) = (m_1 * m_2, n_1 * n_2)$. The unit is the pair of units and the full resource is the pair of full resources.

Sum guard algebra construction. Given guard algebras M and N, the sum construction $M + N$ is the discriminated (or disjoint) union of M and N, up to identifying the units and identifying the full resources of the two PCMs. The PCM operation embeds the operations of each of the constituent PCMs, with composition between elements of different PCMs undefined.

Within the Caper implementation, guards are represented as maps from guard names to parameters that depend on the guard type. For this to work, Caper disallows multiple guards with the same name in a guard algebra definition. For instance, `INSERT * %INSERT` is not legal. The sum construction is implemented by rewriting where necessary by the identities the construction introduces.

3.2 Interference Reasoning

There are two sides to interference: on one side, a thread should only perform actions that are anticipated by the environment, expressed by the Guar relation; on the other, a thread must anticipate all actions that the environment could perform, expressed by the Rely relation. Each time a thread updates a region, it must ensure that the update is permitted by the Guar with respect to the guards it owns (initially) for that region. Moreover, the symbolic state between operations and frames for non-atomic operations must be *stabilised* by closing the set of states they might be in under the Rely relation. Caper must therefore be able to compute with these relations effectively.

The biggest obstacle to effective computation is that the relations are transitively closed. Transitivity is necessary, at least for Rely, since the environment may take arbitrarily many steps in between the commands of a thread. However, computing the transitive closure in general is a difficult problem. For instance, consider a region that has the (unguarded) action `: n ⇝ n + 1`. From this action, we should infer the relation $\{(n, m) \mid n \leq m\}$, as the reflexive-transitive closure of $\{(n, n+1) \mid n \in \mathbb{Z}\}$. It is generally beyond the ability of SMT solvers to compute transitive closures, although some (limited) approaches have been proposed [14].

Caper employs two techniques to deal with the transitive closure problem. This first is that, if the state space of the region is finite, then it is possible to compute the transitive closure directly. Caper uses a variant of the Floyd-Warshall algorithm [16] for computing shortest paths in a weighted graph. The 'weights' are constraints (first-order formulae) with conjunction as the 'addition' operation and disjunction as the 'minimum' operation.

The second technique is to add composed actions until the set of actions is transitively closed. When the actions are transitively closed, the Rely and Guar relations can be computed without further accounting for transitivity (i.e. the (†) rules in Fig. 5 can be ignored). For two actions $P \mid G : e_1 \rightsquigarrow e_2$ and $P' \mid G' : e'_1 \rightsquigarrow e'_2$ (assuming the only common variables are region parameters) their composition is $P, P', e_2 = e'_1 \mid G \sqcup G' : e_1 \rightsquigarrow e'_2$ where $\sqcup$ is the least-upper-bound operation on guards. Using frame inference for guards, we can check if one action subsumes another — that is, whether any transition permitted by the second is also permitted by the first.

Caper uses the following process to reach a transitive set of actions. First, consider the composition of each pair of actions currently in the set and determine if it is subsumed by any action in the set. If all compositions are already subsumed then the set is transitive. Otherwise, add all compositions that are not subsumed and repeat. Since this process is not guaranteed to terminate (for example, for `n ⇝ n + 1`), Caper will give up after a fixed number of iterations fail to reach a transitive set. Note that adding composite actions does not change the Rely and Guar relations, since these are defined to be transitively closed.

If Caper is unable to reach a transitive set of actions, the Rely relation is over-approximated by the universal relation, while the Guar is under-approximated. This is sound, since Caper can prove strictly less in such circumstances, although the over-approximation is generally too much to prove many examples.

It is often practical to represent the transition system for a region type in a way that Caper can determine its transitive closure. For example, instead of the action `: n ⇝ n + 1` we can give the action `n < m | : n ⇝ m`, which Caper can prove subsumes composition with itself. Since Caper tries to find a transitive closure, it is often unnecessary to provide a set of actions that is transitively closed. For instance, given the actions `0 ≤ n, n < m | A : n ⇝ -m` and `0 < n | B : -n ⇝ n`, Caper adds the following actions to reach a transitive closure:

```
0 ≤ n, n < m | A * B : n ⇝ m;      0 < n, n < m | A * B : -n ⇝ -m;
0 < n, n < m | A * B : -n ⇝ m;
```

3.3 Symbolic Execution

Caper's proof system is based on symbolic execution, where programs are interpreted over a domain of symbolic states. Symbolic states represent separation logic assertions, but are distinct from the syntactic assertions of the Caper input language. To verify that code satisfies a specification, the code is symbolically executed from a symbolic state corresponding to the precondition. The symbolic execution may be non-deterministic — for instance, to account for conditional statements — and so produces a set of resulting symbolic states. If each of these symbolic states entails the postcondition, then the code satisfies the specification.

A symbolic state $S = (\mathit{\Delta}, \mathit{\Pi}, \mathit{\Sigma}, \mathit{\Xi}, \mathit{\Upsilon}) \in \mathsf{SState}$ consists of:

- $\mathit{\Delta} \in \mathsf{VarCtx} = \mathsf{SVar} \rightharpoonup_{\mathrm{fin}} \mathsf{Sort}$, a *variable context* associating logical sorts with symbolic variables;
- $\mathit{\Pi} \in \mathsf{Pure} = \mathsf{Cond}^*$, a context of *pure conditions* (over the symbolic variables) representing logical assumptions;
- $\mathit{\Sigma} \in \mathsf{Preds} = \mathsf{Pred}^*$, a context of *predicates* (over the symbolic variables) representing owned resources;
- $\mathit{\Xi} \in \mathsf{Regions} = \mathsf{RId} \rightharpoonup_{\mathrm{fin}} \mathsf{RType}_\bot \times \mathsf{Exp}_\bot \times \mathsf{Guard}$, a finite map of region identifiers to an (optional) region type, an (optional) expression representing the state of the region, and an guard expression representing the owned guards for the region;
- $\mathit{\Upsilon} \in \mathsf{ProgVars} = \mathsf{ProgVar} \rightharpoonup_{\mathrm{fin}} \mathsf{Exp}$, a map from program variables to expressions representing their current values.

$$\frac{\mathrm{dom}(\gamma) = S \qquad \mathrm{range}(\gamma) = \mathit{\Gamma} \qquad \forall x, y \in S.\, \gamma(x) = \gamma(y) \implies x = y \qquad \mathit{\Delta} \cap \mathit{\Gamma} = \emptyset}{\mathit{freshSub}(\mathit{\Delta}, \mathit{\Gamma}, S, \gamma)}$$

$$\frac{\begin{array}{c} s'_1, \dots, s'_n \notin \mathit{\Delta} \qquad \mathit{\Xi} = [r_1 \mapsto (t_1, s_1, G_1), \dots, r_n \mapsto (t_n, s_n, G_n)] \\ P = (s_1, s'_1) \in \mathrm{Rely}(t_1, G_1); \dots ; (s_n, s'_n) \in \mathrm{Rely}(t_n, G_n) \\ \mathit{\Xi}' = [r_1 \mapsto (t_1, s'_1, G_1), \dots, r_n \mapsto (t_n, s'_n, G_n)] \end{array}}{\mathit{stabilise}(\mathit{\Delta}, \mathit{\Xi}, s'_1 : \mathbf{Val}; \dots ; s'_n : \mathbf{Val}, P, \mathit{\Xi}')}$$

$$\frac{\begin{array}{c} \mathit{\Phi}(f) = (\bar{x}, A_{\mathrm{pre}}, A_{\mathrm{post}}) \\ \mathit{freshSub}(\varepsilon, \mathit{\Delta}, \mathrm{vars}(A_{\mathrm{pre}}) \cup \bar{x}, \gamma) \qquad \mathit{produce}(A_{\mathrm{pre}}, \gamma) : (\mathit{\Delta}, \varepsilon, \varepsilon, \emptyset) \rightsquigarrow \mathbb{S}_0 \\ \forall (\mathit{\Delta}_0, \mathit{\Pi}_0, \mathit{\Sigma}_0, \mathit{\Xi}_0) \in \mathbb{S}_0.\, \exists \mathbb{S}_1.\, \vdash C : (\mathit{\Delta}_0, \mathit{\Pi}_0, \mathit{\Sigma}_0, \mathit{\Xi}_0, [\bar{x} \mapsto \gamma(\bar{x})]) \rightsquigarrow \mathbb{S}_1 \\ \forall (\mathit{\Delta}_1, \mathit{\Pi}_1, \mathit{\Sigma}_1, \mathit{\Xi}_1, \mathit{\Upsilon}_1) \in \mathbb{S}_1.\, \exists \mathit{\Delta}'_1, \mathit{\Pi}'_1, \mathit{\Xi}'_1.\, \mathit{stabilise}(\mathit{\Delta}_1, \mathit{\Xi}_1, \mathit{\Delta}'_1, \mathit{\Pi}'_1, \mathit{\Xi}'_1) \\ \exists \mathit{\Gamma}_1, \gamma'.\, \mathit{freshSub}(\mathit{\Delta}_1; \mathit{\Delta}'_1, \mathit{\Gamma}_1, \mathrm{vars}(A_{\mathrm{post}}) \setminus (\mathrm{vars}(A_{\mathrm{pre}}) \cup \bar{x}), \gamma') \\ \exists \mathbb{S}_2.\, \mathit{consume}(A_{\mathrm{post}}, \gamma \cup \gamma') : (\mathit{\Delta}_1; \mathit{\Delta}'_1, \mathit{\Pi}_1; \mathit{\Pi}'_1, \mathit{\Gamma}_1, \varepsilon, \mathit{\Sigma}_1, \mathit{\Xi}'_1) \rightsquigarrow \mathbb{S}_2 \\ \forall (\mathit{\Delta}_2, \mathit{\Pi}_2, \mathit{\Gamma}_2, P_2, \mathit{\Sigma}_2, \mathit{\Xi}_2) \in \mathbb{S}_2.\, \mathit{\Delta}_2, \mathit{\Pi}_2 \vdash \mathit{\Gamma}_2, P_2 \end{array}}{\mathit{\Psi}, \mathit{\Phi} \vdash \mathtt{function}\ f(\bar{x})\{C\}}$$

Fig. 6. Function correctness judgement.

We take the set of symbolic variables SVar to be countably infinite. Symbolic variables are considered distinct from program variables (ProgVar), which occur in the syntax of the program code (Stmt), and assertion variables (AssnVar), which occur in syntactic assertions (Assn). Currently, the set of sorts supported by Caper is Sort $= \{\mathbf{Val}, \mathbf{Perm}, \mathbf{RId}\}$. That is, a variable can represent a program value (i.e. an integer), a permission, or a region identifier.

We do not formally define the syntax of (symbolic) expressions (Exp) and conditions (Cond). Expressions include symbolic variables, as well as arithmetic operators and operators on permissions. Conditions include a number of relational propositions over expressions, such as equality and inequality ($<$). They can also express rely and guarantee relations and inclusions between sets. A context of pure conditions is a sequence of conditions, interpreted as a conjunction. Symbolic execution generates entailments between contexts of conditions as verification conditions. The practical limitation on conditions is that these entailments should be checkable automatically by means of provers such as SMT solvers.

A (spatial) predicate $P(\bar{e}) \in$ Pred consists of a predicate name P and a list of expressions $\bar{e}$. Two types of predicates are given special treatment and have their own syntax: individual heap cells $a \mapsto b$ (where a is the address and b the value stored), and blocks of heap cells $a \mapsto \#\mathrm{cells}(n)$ (where a is the starting address and n is the number of consecutive heap cells). All other predicates are abstract.

A region map associates region identifiers with knowledge and resources for the given region. The knowledge consists of the type of the region, which is a pair of a region type name and list of expressions representing the parameters, and an expression describing the current state of the region. The resources consist of a guard. It is possible to have a guard for a region without knowing the type or state of the region, so these two components can be unspecified ($\bot$).

Fig. 6 gives the correctness judgement for functions that forms the basis of Caper's proof system. The judgement is parametrised by a context of region declarations $\mathit{\Psi}$, and a context of function specifications $\mathit{\Phi}$. (Both contexts are left implicit in the sub-judgements.) The conditions break down into four key steps:

1. A symbolic state is generated corresponding to the function's precondition A_{pre}. This is captured by the judgement $\mathit{produce}(A_{\mathrm{pre}}, \gamma) : (\mathit{\Delta}, \varepsilon, \varepsilon, \emptyset) \rightsquigarrow \mathbb{S}_0$ which adds resources and assumptions to an initially empty symbolic state.
2. The body of the function is symbolically executed. This is captured by the judgement $\vdash C : (\mathit{\Delta}_0, \mathit{\Pi}_0, \mathit{\Sigma}_0, \mathit{\Xi}_0, [\bar{x} \mapsto \gamma(\bar{x})]) \rightsquigarrow \mathbb{S}_1$.
3. The regions of the resulting symbolic states are stabilised to account for possible interference from other threads. This is captured by the judgement $\mathit{stabilise}(\mathit{\Delta}_1, \mathit{\Xi}_1, \mathit{\Delta}_1', \mathit{\Pi}_1', \mathit{\Xi}_1')$. (Note that stabilisation also occurs at each interleaving step in the symbolic execution.)
4. Each final symbolic state is checked against the postcondition. This is captured by the judgement $\mathit{consume}(A_{\mathrm{post}}, \gamma \cup \gamma') : (\mathit{\Delta}_1, \mathit{\Pi}_1, \mathit{\Gamma}, \varepsilon, \mathit{\Sigma}_1, \mathit{\Xi}_1) \rightsquigarrow \mathbb{S}_2$ which removes resources and generates verification conditions that are sufficient for the symbolic state to entail the postcondition. These verification conditions are checked by the judgement $\mathit{\Delta}_2, \mathit{\Pi}_2 \vdash \mathit{\Gamma}_2, P_2$.

Step 1 uses the judgement $\mathit{produce} \subseteq (\mathsf{Assn} \times (\mathsf{AssnVar} \rightharpoonup \mathsf{Exp})) \times \overline{\mathsf{SState}} \times \mathcal{P}(\overline{\mathsf{SState}})$, where $\overline{\mathsf{SState}} = \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{Preds} \times \mathsf{Regions}$. The *produce* judgement (we adopt the produce/consume nomenclature of Verifast [18]) adds resources and assumptions to the symbolic state corresponding to a given syntactic assertion. It is parametrised by a substitution from assertion variables to expressions. In producing the precondition, this substitution maps the assertion variables occurring in the precondition and the function parameters (treated as assertion variables) to fresh symbolic variables. This is captured by the *freshSub* judgement. These fresh symbolic variables are bound in the initial variable context ($\mathit{\Delta}$), while the initial context of conditions (ε), context of predicates (ε) and region map ($\emptyset$) are all empty. The judgement produces a set of symbolic states (sans program variable context). This set should be interpreted disjunctively: each of the symbolic states is possible after producing the assertion.

Step 2 uses the symbolic execution judgement: $(\vdash - : - \rightsquigarrow -) \subseteq \mathsf{Stmt} \times \mathsf{SState} \times \mathcal{P}(\mathsf{SState})$. This judgement updates the symbolic state according to the symbolic execution rules for program statements. The initial program variable context is given by mapping the function parameters $\bar{x}$ (treated as program variables) to the corresponding logical expressions $\gamma(\bar{x})$.

Step 3 uses the judgement $\mathit{stabilise} \subseteq \mathsf{VarCtx} \times \mathsf{Regions} \times \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{Regions}$. This judgement relates an initial region map (in a given context) with a new region map that accounts for interference, with an extended context and additional pure conditions. The judgement is defined by the rule given in Fig. 6. This rule creates a fresh variable (s_i') to represent the new state of each region and asserts that it is related to the old state (s_i) in accordance with the rely relation for the given region. To account for the region type or state being unknown, we extend the definition of Rely with the following two rules:

$$\frac{}{(x, y) \in \mathrm{Rely}(\bot, G)} \qquad \frac{}{(\bot, y) \in \mathrm{Rely}(\mathbf{T}(r, \bar{x}), G)}$$

Step 4 uses the judgement *consume* $\subseteq (\mathsf{Assn} \times (\mathsf{AssnVar} \rightharpoonup \mathsf{Exp})) \times \widehat{\mathsf{SState}} \times \mathcal{P}(\widehat{\mathsf{SState}})$, where $\widehat{\mathsf{SState}} = \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{Preds} \times \mathsf{Regions}$. The *consume* judgement removes resources and adds assertions to the symbolic state. The symbolic state is extended with a second variable context representing existentially quantified variables and a second context of pure conditions representing logical assertions. As an example, consuming the assertion `x |-> 2` where the predicates include $a \mapsto b$ can remove that predicate, adding the assertions $[\![\mathtt{x}]\!]_\gamma = a$ and $2 = b$ (where γ is the assertion variable substitution). Any assertion variables occurring in the postcondition that are neither parameters of the function nor occur in the precondition are treated as existentially quantified. The *freshSub* judgement is used again to generate a context and substitution for these variables.

It remains to check that the assertions arising from consuming the postcondition follow from the assumptions. This is achieved with the entailment judgement: $(-, - \vdash -, -) \subseteq \mathsf{VarCtx} \times \mathsf{Pure} \times \mathsf{VarCtx} \times \mathsf{Pure}$. The judgement is defined by:

$$\varDelta, \varPi \vdash \varGamma, P \quad \overset{\text{def}}{\iff} \quad \forall \delta \in [\![\varDelta]\!] \,.\, [\![\varPi]\!]_\delta \implies \exists \delta' \in [\![\varGamma]\!] \,.\, [\![P]\!]_{\delta \cup \delta'}$$

Here, $[\![\varDelta]\!]$ is the set of variable assignments agreeing with context $\varDelta$ and $[\![\varPi]\!]_\delta$ is the valuation of the conjunction of the conditions $\varPi$ in the variable assignment δ.

The produce judgement. The rules for the *produce* judgement are given in Fig. 7. The rules follow the syntax of the assertion to be produced. For a separating conjunction (`&*&`), first the left assertion is produced and then the right. Producing a conditional expression (`?:`) introduces non-determinism: we generate cases for whether the condition is true or false. For the true case, the first assertion is produced together with the condition; for the false case, the second assertion is produced together with the negated condition. Note that this non-determinism is demonic, in that the proof must deal with all cases. Producing a pure assertion simply adds it to the logical assumptions (interpreting the assertion variables through the substitution γ). Producing a predicate assertion adds it to the predicate context. As a special case, the points-to predicate also adds logical assumptions, expressed by $\mathcal{C}_{\text{cell}}$, which express that addresses must be positive and no two cells can have the same address (we elide the formal definition here).

The remaining two rules concern regions: producing a region and a guard assertion respectively. In each case, a region descriptor r_0 is created — in the first case, including the region type and state but the empty guard, and in the second case, including a guard but no region type or state. We non-deterministically consider two cases: when the region identified by z already exists in the symbolic state, and when it represents a completely fresh region. The first case is handled by rmerge, which non-deterministically merges the region with one of the existing regions. The second case is handled by rnew, which associates the region with a fresh identifier (i) and adds assumptions that this is distinct from all other identifiers. Merging regions is governed by the *mergeRegion* judgement, which combines two region descriptors into one, producing a series of assumptions that are necessary for the merger to be well-defined. We elide the details here.

$$\frac{\begin{array}{c}\mathit{produce}(A_1, \gamma) : S \rightsquigarrow \{S_i\}_{i \in I} \\ \forall i \in I.\, \mathit{produce}(A_2, \gamma) : S_i \rightsquigarrow \{S_{i,j}\}_{j \in J_i}\end{array}}{\mathit{produce}(A_1 \;\texttt{\&*\&}\; A_2, \gamma) : S \rightsquigarrow \left\{S_{i,j} \;\middle|\; i \in I, j \in J_i\right\}}$$

$$\frac{\begin{array}{c}\mathit{produce}(A_1, \gamma) : (\Delta, \Pi; \llbracket p \rrbracket_\gamma, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \\ \mathit{produce}(A_2, \gamma) : (\Delta, \Pi; \neg \llbracket p \rrbracket_\gamma, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_2\end{array}}{\mathit{produce}(p \mathbin{?} A_1 : A_2, \gamma) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\frac{}{\mathit{produce}(p, \gamma) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow (\Delta, \Pi; \llbracket p \rrbracket_\gamma, \Sigma, \Xi)}$$

$$\frac{}{\mathit{produce}(P(\bar{e}), \gamma) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow (\Delta, \Pi, \Sigma; P(\llbracket \bar{e} \rrbracket_\gamma), \Xi)}$$

$$\frac{}{\mathit{produce}(e_1 \mapsto e_2, \gamma) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow (\Delta, \Pi; \mathcal{C}_{\mathrm{cell}}(\llbracket e_1 \rrbracket_\gamma, \Sigma), \Sigma; \llbracket e_1 \rrbracket_\gamma \mapsto \llbracket e_2 \rrbracket_\gamma, \Xi)}$$

$$\frac{\begin{array}{c}r_0 = (\mathbf{T}(\llbracket z, \bar{e} \rrbracket_\gamma), \llbracket s \rrbracket_\gamma, 0) \qquad \mathbb{S}_1 = \mathrm{rmerge}(\Delta, \Pi, \Sigma, \Xi, \llbracket z \rrbracket_\gamma, r_0) \\ i \notin \mathrm{dom}(\Xi) \qquad \mathbb{S}_2 = \mathrm{rnew}(\Delta, \Pi, \Sigma, \Xi, \llbracket z \rrbracket_\gamma, r_0, i)\end{array}}{\mathit{produce}(\mathbf{T}(z, \bar{e}, s), \gamma) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\frac{\begin{array}{c}r_0 = (\bot, \bot, \llbracket G \rrbracket_\gamma) \qquad \mathbb{S}_1 = \mathrm{rmerge}(\Delta, \Pi, \Sigma, \Xi, \llbracket z \rrbracket_\gamma, r_0) \\ i \notin \mathrm{dom}(\Xi) \qquad \mathbb{S}_2 = \mathrm{rnew}(\Delta, \Pi, \Sigma, \Xi, \llbracket z \rrbracket_\gamma, r_0, i)\end{array}}{\mathit{produce}(z@(G), \gamma) : (\Delta, \Pi, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\begin{array}{l}\mathrm{rmerge}(\Delta, \Pi, \Sigma, \Xi, a, r_0) = \\ \qquad \left\{(\Delta, \Pi; a = i'; \Pi', \Sigma, \Xi[i' \mapsto r']) \;\middle|\; \exists r.\, \Xi(i) = r \wedge \mathit{mergeRegion}(r, r_0, \Pi', r')\right\} \\ \mathrm{rnew}(\Delta, \Pi, \Sigma, \Xi, a, r_0, i) = \left\{\left(\Delta, \Pi; a = i; \bigwedge \left\{i \neq i' \;\middle|\; i' \in \mathrm{dom}(\Xi)\right\}, \Sigma, \Xi[i \mapsto r_0]\right)\right\}\end{array}$$

Fig. 7. Selected rules for the *produce* judgement.

The consume judgement. A selection of rules for the *consume* judgement are given in Fig. 8. Unlike with *produce*, the syntax of the assertion may not uniquely determine which rule to apply. For instance, there are two rules for consuming a conditional assertion. The first rule consumes the conditional by consuming either the condition and the first assertion or the negated condition and the second assertion. (This are somewhat analogous to the $\vee$-introduction rules of natural deduction.) The second rule non-deterministically *assumes* the truth or falsity of the condition, consuming the first or second assertion in the respective case. This requires that only assumption variables can occur in the condition ($\mathrm{fv}(\llbracket p \rrbracket_\gamma) \subseteq \Delta$), since otherwise the context of assumptions would be ill-formed. Here, we are exploiting the law of the excluded middle for pure assertion p: that is $p \vee \neg p$ holds. Consuming a region assertion asserts that there is a corresponding region with the specified type and state. Consuming a guard assertion makes use of the judgement $\mathit{takeGuard}(\Delta, t, H, G, \Gamma, P, F)$, which expresses that guard G can be removed from guard H leaving the frame F, under conditions P, given the region type t. Frame inference is used to discharge *takeGuard* obligations.

The symbolic execution judgement. The symbolic execution judgement expresses how executing a program statement affects the symbolic state. Most of the sym-

$$\frac{(A, q) \in \{(A_1, \llbracket p \rrbracket_\gamma), (A_2, \neg \llbracket p \rrbracket_\gamma)\} \qquad \mathit{consume}(A, \gamma) : (\Delta, \Pi, \Gamma, P; q, \Sigma, \Xi) \rightsquigarrow \mathbb{S}}{\mathit{consume}(p \mathbin{?} A_1 : A_2, \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}}$$

$$\frac{\begin{array}{c}\mathrm{fv}(\llbracket p \rrbracket_\gamma) \subseteq \Delta \qquad \mathit{consume}(A_1, \gamma) : (\Delta, \Pi; \llbracket p \rrbracket_\gamma, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \\ \mathit{consume}(A_2, \gamma) : (\Delta, \Pi; \neg \llbracket p \rrbracket_\gamma, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_2\end{array}}{\mathit{consume}(p \mathbin{?} A_1 : A_2, \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \rightsquigarrow \mathbb{S}_1 \cup \mathbb{S}_2}$$

$$\frac{\Xi(i) = ((R, \bar{x}), s, G) \qquad s \neq \bot}{\begin{array}{c}\mathit{consume}(R(r, \bar{e}, a), \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \\ \rightsquigarrow \{(\Delta, \Pi, \Gamma, P; \llbracket r \rrbracket_\gamma = i; \llbracket r, \bar{e} \rrbracket_\gamma = \bar{x}; \llbracket a \rrbracket_\gamma = s, \Sigma, \Xi)\}\end{array}}$$

$$\frac{\Xi(i) = (t, s, H) \qquad \mathit{takeGuard}(\Delta, t, H, \llbracket G \rrbracket_\gamma, \Gamma', F, P')}{\begin{array}{c}\mathit{consume}(r@G, \gamma) : (\Delta, \Pi, \Gamma, P, \Sigma, \Xi) \\ \rightsquigarrow \big\{(\Delta, \Pi, \Gamma; \Gamma', P; P', \Sigma, \Xi[i \mapsto (t, s, F)])\big\}\end{array}}$$

Fig. 8. Selected rules for the *consume* judgement.

bolic execution rules are standard, except that when statements are sequenced together, the intermediate symbolic state is stabilised (using the *stabilise* judgement). The other novelty is the symbolic execution of atomic statements (read, write and CAS operations), which require access to the shared regions. The symbolic execution rule is given in Fig. 9. It consists of six steps:

1. Regions are opened with the *openRegions* judgement. This is based on the *open* judgement, which opens a single region by producing its interpretation (by case analysis on the possible interpretations).
2. The atomic statement is symbolically executed with the *atomic* judgement. This cannot affect the shared regions.
3. The regions are updated with the *updateRegions* judgement. This applies the *update* judgement to each of the regions, updating the state arbitrarily in a manner that is consistent with the guarantee for the available guard.
4. New regions may be created with the *createRegions* judgement. These regions must be distinct from the existing ones, and will be created along with the full guard for the region type. At this point, these new regions are open.
5. All of the open regions are closed with the *closeRegions* judgement. This applies *close* for each region, which consumes the interpretation.
6. The generated assertions are checked to follow from the assumptions, and the assertions are treated as assumptions in the new symbolic state.

4 Proof Search

The success of separation logic as a basis for symbolic execution is in part due to its determinacy: given a precondition and a program statement, it is feasible to compute the strongest postcondition or determine that the program might fault. Concurrent separation logics, however, depend on auxiliary state, such as Caper's regions and guards, which can introduce non-determinism since the program itself does not specify how the auxiliary state should be updated. For instance,

$$\frac{\begin{array}{c} \varXi(r) = (\mathbf{T}(\bar{x}), s, G) \\ \varPsi_{\mathrm{I}}(\mathbf{T}) = \{(P_i, e_i, A_i)\}_{i \in I} \qquad \forall i \in I.\, \mathit{freshSub}(\varDelta, \varGamma_i, \mathrm{vars}(P_i, e_i, A_i), \gamma_i) \\ \forall i \in I.\, \mathit{produce}(A_i, \gamma_i) : (\varDelta; \varGamma_i, \varPi; [\![\varPsi_{\mathrm{P}}(\mathbf{T})]\!]_{\gamma_i} = \bar{x}; s = [\![e_i]\!]_{\gamma_i}; [\![P_i]\!]_{\gamma_i}, \varSigma, \varXi) \rightsquigarrow \mathbb{S}_i \end{array}}{\mathit{open}(r) : (\varDelta, \varPi, \varSigma, \varXi) \rightsquigarrow \bigcup \{\mathbb{S}_i\}_{i \in I}}$$

$$\frac{\varXi(r) = (\mathbf{T}(\bar{x}), s, G) \qquad s' \notin \varDelta \qquad \varGamma = (s' : \mathbf{Val}) \qquad P = ((s, s') \in \mathrm{Guar}_{\varPsi}(\mathbf{T}(\bar{x}), G))}{\mathit{update}(r, \varDelta, \varXi, \varGamma, P, \varXi[r \mapsto (\mathbf{T}(\bar{x}), s', G)])}$$

$$\frac{\begin{array}{c} \varXi(r) = (\mathbf{T}(\bar{x}), s, G) \\ (P_0, e_0, A_0) \in \varPsi_{\mathrm{I}}(\mathbf{T}) \qquad \mathit{freshSub}(\varDelta; \varGamma, \varGamma_0, \mathrm{vars}(P_0, e_0, A_0), \gamma) \\ \mathit{consume}(A_0, \gamma) : (\varDelta, \varPi, \varGamma; \varGamma_0, P; [\![\varPsi_{\mathrm{P}}(\mathbf{T})]\!]_{\gamma} = \bar{x}; s = [\![e_0]\!]_{\gamma}; [\![P_0]\!]_{\gamma}, \varSigma, \varXi) \rightsquigarrow \mathbb{S} \end{array}}{\mathit{close}(r) : (\varDelta, \varPi, \varGamma, P, \varSigma, \varXi) \rightsquigarrow \mathbb{S}}$$

$$\frac{\begin{array}{c} \mathit{openRegions} : (\varDelta, \varPi, \varSigma, \varXi, \varepsilon) \rightsquigarrow \mathbb{S} \\ \forall (\varDelta_1, \varPi_1, \varSigma_1, \varXi_1, \bar{r}) \in \mathbb{S}.\, \exists \mathbb{S}_1.\, \mathit{atomic}(\alpha) : (\varDelta_1, \varPi_1, \varSigma_1, \varUpsilon) \rightsquigarrow \mathbb{S}_1 \\ \forall (\varDelta_2, \varPi_2, \varSigma_2, \varUpsilon_2) \in \mathbb{S}_1.\, \exists \varGamma_2, P_2, \varXi_2.\, \mathit{updateRegions}(\bar{r}, \varDelta_2, \varXi_1, \varGamma_2, P_2, \varXi_2) \\ \exists \bar{s}, \varGamma_2', P_2', \varXi_2'.\, \mathit{createRegions}(\bar{s}, \varDelta_2; \varGamma_2, \varXi_2, \varGamma_2', P_2', \varXi_2') \\ \exists \mathbb{S}_2.\, \mathit{closeRegions}(\bar{s}, \bar{r}) : (\varDelta_2, \varPi_2, \varGamma_2; \varGamma_2', P_2; P_2', \varSigma_2, \varXi_2') \rightsquigarrow \mathbb{S}_2 \\ \forall (\varDelta_3, \varPi_3, \varGamma_3, P_3, \varSigma_3, \varXi_3) \in \mathbb{S}_3.\, (\varDelta_3, \varPi_3 \vdash \varGamma_3, P_3) \land (\varDelta_3; \varGamma_3, \varPi_3; P_3, \varSigma_3, \varXi_3, \varUpsilon_2) \in \mathbb{S}_3 \end{array}}{\vdash \langle \alpha \rangle : (\varDelta, \varPi, \varSigma, \varXi, \varUpsilon) \rightsquigarrow \mathbb{S}_3}$$

Fig. 9. Symbolic execution rule for atomic statements.

when closing a region, Caper may have more than one valid choice for the region state. Consequently, non-determinism is fundamental to Caper's design and backtracking is used to resolve the non-deterministic choices in the proof search. This non-determinism is used in: consuming conditional or disjunctive assertions; determining which heap cell, region or predicate to consume; choosing which regions to open; determining which state interpretation to close a region with; and determining how to rewrite guards.

Since non-determinism introduces branching, which can be detrimental to performance, Caper uses some heuristics in an effort to prune or avoid bad branches. For instance, when consuming a heap cell, if there is a cell whose address matches syntactically (e.g. consuming `x |-> z` in a symbolic state with `x |-> 2`) Caper will not consider other cases, which would be eliminated by a later SMT call. Another example is that Caper prioritises opening *different* regions over opening *more* regions. Opening a region typically involves case analysis, and so opening multiple regions can lead to bad performance when it is not required.

It is important to note that choices by the user can significantly affect Caper's level of non-determinism. For instance, if the abstract state of a region is determined exactly by the concrete state, Caper will consider fewer possibilities than if a region can be in multiple states for the same concrete state.

There are a some non-deterministic choices that should not generally be considered as options everywhere they could be allowed. One of these is region creation: at any point it is legal to create a new region, provided the appropriate resources are available. It would quickly make proof search intractable if Caper considered all possible ways of creating regions at all times. On the other hand, there is a good indicator for when creating a region will be helpful: when Caper fails to consume a region assertion. Caper's backtracking mechanism includes handlers that introduce additional non-determinism in response to specific failures. In particular, a failure to consume a region assertion will be handled by attempting to create an appropriate region.

This behaviour is a form of abductive reasoning: the general reasoning principle of inferring explanatory hypotheses from a goal. It has previously been applied to automatically derive separation logic specifications [4]. Our approach differs in that we infer missing updates to auxiliary state rather than missing resources in the precondition.

A further application of abduction in Caper is in handling existential logical variables. Consuming an assertion may introduce a new (existential) logical variable with some constraints. Caper calls the prover to check that these constraints are satisfiable; thereafter, Caper only knows that the variable satisfies these constraints. If, later in the symbolic execution, Caper requires the variable to satisfy additional constraints, then the proof may fail even if there was a witness satisfying these constraints. The constraints are therefore abductively propagated back to where the variable was introduced, and the proof is retried. Since Caper then knows that the additional constraints are satisfied, it can successfully discharge them when they arise.

5 Evaluation

We have successfully applied Caper to verify a number of concurrent algorithms. In §2, we discussed the spin lock and ticket lock, whose specifications guarantee mutual exclusion. We have verified a reader-writer lock, whose specification permits multiple readers or a single writer to enter their critical sections concurrently. This example uses counting permissions, but we have also verified a bounded version that does not. We have also verified a number of counter implementations with specifications that enforce monotonicity (CASCounter, BoundedCounter and IncDec). We have verified a library for joining on forked threads and a client that waits for the child thread to terminate before presenting the work done by the child. We have also verified a synchronisation barrier and a client that uses it to synchronise threads incrementing and decrementing a counter. Finally, we have verified two implementations of a bag: the stack of §2.3, and a concurrent queue. We summarise these examples in Table 1, which shows the number of lines of code and annotation and verification times for recursive and iterative versions of each example. By default, Caper can create up to two regions at a time (`-c 2`) and open up to two regions (`-o 2`). The BagStack and Queue examples require opening up to three regions. The BarrierClient requires creating no regions, because of an issue with Caper's failure handling implementation.

From the verification times, we can observe that the versions that use loops tend to take longer than the recursive versions. This is due to case analysis which propagates through loops, but is abstracted in function calls. The high verification times for the bounded examples are largely due to Caper computing transitive

Name	Code (lines)	Annotations (lines)	Time (s)
SpinLock	17 / 17	17 / 18	0.23 / 0.34
TicketLock(Client)	33 (41) / 24 (32)	19 (29) / 17 (27)	0.84 (0.89) / 2.43 (2.45)
ReadWriteLock	36 / 37	25 / 28	3.33 / 15.86
BoundedReadWriteLock	55 / 57	36 / 41	30.68 / 126.04
CASCounter	20 / 20	15 / 16	0.08 / 0.15
BoundedCounter	25 / 25	20 / 21	3.95 / 19.88
IncDec	29 / 29	19 / 21	0.10 / 0.36
ForkJoin(Client)	17 (32) / 17 (32)	16 (30) / 17 (31)	0.06 (0.07) / 0.07 (0.09)
Barrier(Client*)	71 (127) / 77 (130)	31 (60) / 35 (67)	21.11 (29.80) / 26.96 (31.84)
BagStack†	35 / 30	26 / 26	3.18 / 11.76
Queue‡	60 / 58	37 / 38	177.38 / 179.61

* flags: `-c 0` † flags: `-c 1 -o 3` ‡ flags: `-c 2 -o 3`

Table 1. Examples (recursive/iterative).

closures for finite-state regions. The barrier example also takes significant time to compute the transitive closure of an infinite-state region. The BagStack and Queue use nested regions, and Queue has a complicated transition system, which combine to give a long verification time.

There are three key areas where Caper could use significant improvement. Firstly, proof search could be improved, for instance by directing the choice of regions to open and abstracting multiple branches into one. Currently, successful proofs may take some time and failing proofs take even longer. Secondly, Caper heuristics used in abduction require improvement, including loop invariants, this should allow more algorithms to be proved. Thirdly, Caper's annotations limit the expressivity of specifications to some extent. For instance, there is no support for regions with abstract states other than integers. Despite these limitations, we believe that Caper demonstrates the viability of our approach, and provides a good basis for further investigation.

6 Related Work

Caper is a tool for automating proofs in a concurrent separation logic with shared regions, aimed at proving functional correctness for fine-grained concurrent algorithms. The logic is in the spirit of concurrent abstract predicates (CAP) [10] taking inspiration from recent developments in concurrent separation logic such as iCAP [29], TaDA [7], Views [9], CaReSL [32], FCSL [22] and Iris [19].

Smallfoot [1] pioneered symbolic execution for separation logic. While it can prove functional correctness, it has limited support for concurrency and so cannot prove fine-grained concurrent algorithms.

SmallfootRG [5] extended Smallfoot to the more expressive logic RGSep [33]. The tool uses shared resources that are annotated with invariants and actions that can be performed over these resources. The actions that can be performed are not guarded, which leads to very weak specifications: it can prove memory safety, but not functional correctness. The abstraction of stabilisation employed by SmallfootRG is different than the transitivity-based technique of Caper.

SmallfootRG uses abstract interpretation to weaken assertions such that they are stable, where the abstract domain is based on symbolic assertions. Requiring (or ensuring) that a set of actions are transitively closed can be seen as an abstraction that terminates in a single step.

CAVE [34] built on SmallfootRG's action inference to prove linearisability [17] of concurrent data structures. That is, CAVE can prove that the operations of a concurrent data structure are atomic with respect to each other, and satisfy an abstract functional specification. Caper cannot yet prove linearisability, although it could in future support abstract atomicity in the style of TaDA [7]. On the other hand, CAVE cannot prove functional correctness of non-linearisable examples such as a spin lock.

Other mechanised — but not automatic — approaches based on separation logic include Verifast [18] and the Coq mechanisation of fine-grained concurrent separation logic [22,28]. Both approaches support an expressive assertion language, including higher-order predicates. They are able to prove functional correctness properties for fine-grained concurrent programs. Direct comparisons are, however, difficult. Programs and specifications need adaption, often more than simple translation, resulting in different and sometimes weaker specifications. This is due in part to a smaller core set of operations and in part to a lack of features and expressivity of logic. However, when examples are comparable, the annotation overhead of the Caper examples are lower, often significantly. For example, the spin lock requires 87 lines of annotation in Verifast, compared to 18 in Caper, while the ticket lock requires 123 lines compared to 17. Verifast takes 0.11s to check each of these examples.

Viper [21] is a verification infrastructure for program verification based on permissions. It supports an expressive permission model that includes fractional permissions and symbolic permissions. It would be interesting to develop a front end for Viper that implements Caper's verification approach. A challenging issue is whether (and how) the non-deterministic proof search can be encoded in Viper's intermediate language.

7 Conclusions

We have presented Caper, the first automatic proof tool for a separation logic with CAP-style shared regions, and discussed the significant innovations that it involves. As a prototype, Caper provides a foundation for exploring the possibilities for automation with such a logic. Support for a number of different features will significantly increase the scope of examples that Caper can handle. We anticipate adding support for the following: additional guard algebra constructions; richer logical data types, such as sets and inductive data types; support for abstract and inductive predicates; and support for separation at the level of abstract states in the spirit of FCSL [22] and CoLoSL [25]. We would like to investigate inferring loop invariants and other annotations. We would like to make Caper more usable by providing proofs and failed proofs in a format that can easily be navigated and interpreted by a user. To this end, Caper already provides an interactive proof mode that allows a user to drive the proof search. This enables exploration of, in particular, failing proofs, which have proven itself valuable in development of the tool and the accompanying examples. A further goal is to put Caper on a rigorous footing by formalising its logic in a proof assistant (such as Coq) and using Caper to generate program proofs that can be checked in the proof assistant or by a verified checker.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: Formal Methods for Components and Objects. pp. 115–137. Springer (2006)
2. Biering, B., Birkedal, L., Torp-Smith, N.: Bi hyperdoctrines and higher-order separation logic. In: ESOP (2005)
3. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL. pp. 259–270 (2005)
4. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 289–300. POPL '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1480881.1480917
5. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: SAS 2007. pp. 233–248. Springer Berlin/Heidelberg (2007)
6. Chang, C.C., Keisler, H.J.: Model Theory. Studies in Logic and the Foundations of Mathematics, Elsevier Science (1990)
7. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A Logic for Time and Data Abstraction. In: ECOOP 2014–Object-Oriented Programming, pp. 207–231. Springer Berlin Heidelberg (2014)
8. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer (2008)
9. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300 (2013)
10. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent Abstract Predicates. In: ECOOP. pp. 504–528 (2010)
11. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J.: Caper (source code), https://github.com/caper-tool/caper
12. Dinsdale-Young, T., da Rocha Pinto, P., Andersen, K.J., Birkedal, L.: Caper, automatic verification with concurrent abstract predicates. Technical Appendix: Program logic (2016), http://cs.au.dk/~kja/papers/caper-esop17/techreport.pdf
13. Dockins, R., Hobor, A., Appel, A.W.: A fresh look at separation algebras and share accounting. In: Hu, Z. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 5904, pp. 161–177. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-10672-9_13
14. El Ghazi, A.A., Taghdiri, M., Herda, M.: First-order transitive closure axiomatization via iterative invariant injections. In: NASA Formal Methods, pp. 143–157. Springer (2015)
15. Ershov, Y.L.: Decidability of the elementary theory of distributive lattices with relative complements and the theory of filters. Algebra i Logika 3, 17–38 (1964)

16. Floyd, R.W.: Algorithm 97: Shortest path. Commun. ACM 5(6), 345– (Jun 1962), http://doi.acm.org/10.1145/367766.368168
17. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
18. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. NASA Formal Methods pp. 41–55 (2011)
19. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: POPL. pp. 637–650 (2015)
20. Le, X.B., Gherghina, C., Hobor, A.: Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings, chap. Decision Procedures over Sophisticated Fractional Permissions, pp. 368–385. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-35182-2_26
21. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Verification, Model Checking, and Abstract Interpretation. pp. 41–62. Springer (2016)
22. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating State Transition Systems for Fine-Grained Concurrent Resources. In: ESOP. pp. 290–310 (2014), http://dx.doi.org/10.1007/978-3-642-54833-8_16
23. O'Hearn, P.W.: Resources, Concurrency and Local Reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (Apr 2007)
24. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL (2005)
25. Raad, A., Villard, J., Gardner, P.: Colosl: Concurrent local subjective logic. In: ESOP. pp. 710–735. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), http://dx.doi.org/10.1007/978-3-662-46669-8_29
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74. IEEE (2002)
27. Schulz, S.: System description: E 1.8. In: Logic for Programming, Artificial Intelligence, and Reasoning, pp. 735–743. Springer Berlin Heidelberg (2013)
28. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized Verification of Fine-grained Concurrent Programs. In: 36th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2015) (2015)
29. Svendsen, K., Birkedal, L.: Impredicative Concurrent Abstract Predicates. In: ESOP. pp. 149–168 (2014)
30. Tarski, A.: Arithmetical classes and types of Boolean algebras. Bulletin of the American Mathematical Society 55, 63 (1949)
31. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center (April 1986)
32. Turon, A., Dreyer, D., Birkedal, L.: Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In: ICFP. pp. 377–390 (2013)
33. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge, Computer Laboratory (2008)
34. Vafeiadis, V.: Automatically proving linearizability. In: Computer Aided Verification. pp. 450–464. Springer (2010)