Programs as Resources in Concurrent Separation Logic

July 2, 2018, at around 13:52

Abstract

There are two great traditions for formally verifying programs: the *proof theoretical* approach, such as Hoare logic; and the *model checking* approach. On the one end, one can write very fine specifications to programs and libraries with the first approach, but this usually requires a significant manual effort. In this paper, we aim at mixing the two: we leverage the TLA+ model checker to prove higher-order specifications of libraries in a concurrent separation logic. First, we propose a proof system to relate an implementation in an effectful concurrent functional language with a specification written in PlusCal, and then we explain how to recover compositional higher-order specifications from safety properties we can establish automatically in PlusCal/TLA+.

1 Introduction

There are two great traditions for formally verifying programs: the *proof theoretical* approach, such as Hoare logic; and the *model checking* approach. On the one end, one can write very fine specifications to programs and libraries with the first approach, but this usually requires a significant manual effort. In this paper, we aim at mixing the two by leveraging the TLA+ model checker to prove higher-order specifications of libraries in a concurrent separation logic. First, we propose a proof system to relate an implementation in an effectful concurrent functional language with a specification written in PlusCal, and then we explain how to recover compositional higher-order specifications from safety properties we can establish automatically in PlusCal/TLA+.

One interesting aspect to this is that, in PlusCal, to verify the correctness of some library code, one proves a safety property on one single program, whose correctness is supposed to imply the correctness of all otherwise correct programs that correctly use this library. For instance, consider the PlusCal specification of a spinlock, in Figure 1. It consists in N threads, each endlessly locking and unlocking a single lock. Here, the client code is represented as two `skip` statement, effectively relying on the fact that the client code does not touch the private state of the lock (here, simply the `b` global variable). This turns out to be precisely the kind of arguments that separation logic was made for!

To make this argument formal, still in the case of this spinlock example, we define three functions in our implementation language, HeapLang, for *creating* a new lock, and for *acquiring* and *releasing* such a lock. The idea is that when a user allocates a lock using `newloc`, in the **logical state** (or *ghost state*) of our concurrent separation logic Iris, we allocate a new instance of the `Spinlock` program, effectively using *programs as resources*. The next step is to establish a kind of simulation between the implementation and the specification, keeping in mind that PlusCal programs can *stutter*, which lets us stay at one of the two `skip` statements when client code is execution, for example.

PlusCal PlusCal is a language to write *algorithms*. It was introduced by Lamport as part of the TLA system. It is meant to be used as a more formal alternative to pseudo-code to describe algorithms, be they sequential, concurrent or distributed. PlusCal is a simple imperative While language, with some special support for parameterized programs (several instances of the same program run in parallel, each knowing their thread ID), and a sort of choice operator (the `with` construct).

2 A very simple example

We begin with a very simple example: a spinlock implementation. Figure 1 shows both the PlusCal specification and the Heaplang implementation. One thing to notice is that this

```
======== PlusCal =========
--algorithm SpinLock {
  variable b = FALSE;
  process (Proc \in 1..N)
  {
      ncs: while (TRUE) {
             skip; \* the noncritical section
             await (b = FALSE);
             b = TRUE;

       cs:   skip \* critical section

             b := FALSE
           }
  }
}

======== HeapLang =========
let newlock () = ref false

let rec acquire l =
   if CAS l false true then ()
   else acquire l

let release l = l := false
```

Figure 1. Spinlock

specification states that there are exaclty N threads running, hence the specifications of the implementation wil only work with any finite number of threads, fixed when newlock is called.

Relating the implementation and the specification We want to link the implementation with the specification in the following way:

$$\{\top\}\ \texttt{newlock ()}\ \{l.\ \exists\gamma, \texttt{islock}(\gamma, l) * \circledast_{n=1}^{N} \boxed{\texttt{pc}_n\texttt{=="ncs"}}^{\gamma}\}$$

$$\{\texttt{islock}(\gamma, l) * \boxed{\texttt{pc}_n\texttt{=="ncs"}}^{\gamma}\}\quad \texttt{acquire l}\quad \{().\ \boxed{\texttt{pc}_n\texttt{=="cs"}}^{\gamma}\}$$
$$\{\texttt{islock}(\gamma, l) * \boxed{\texttt{pc}_n\texttt{=="cs"}}^{\gamma}\}\quad \texttt{release l}\quad \{().\ \boxed{\texttt{pc}_n\texttt{=="ncs"}}^{\gamma}\}$$

Here, $\texttt{islock}(\gamma, l)$ is defined as:

$$\boxed{\exists v.\ l \mapsto v * \boxed{\texttt{b==}v}^{\gamma}} * \boxed{\exists \rho, \rho_0 \rightarrow^* \rho * \boxed{\bullet\, \rho}^{\gamma}}$$

where ρ_0 is the initial state of the PlusCal program. In the proof of the Hoare triple for acquire, it is at the execution of a successful CAS instruction that we advance the specification program from the label "ncs" to "cs".

Metaremark 1. *This step is supposed to be as trivial and uninteresting as possible. Indeed, this step is done by hand, and the goal here is to leverage the automation that TLA+ gives us. Typically, in the model checking community, this step would be automated, by essentially generating the PlusCal program from the HeapLang program. This is the main* social *reason to derive the compositional specification.*

All throughout this paper, we will write ρ for the configurations of the specification programs, and σ for that of the Heaplang programs. In order to exploit the safety properties about the PlusCal programs, we need to be able to relate the current configuration with the starting configuration. Indeed, some predicate of the specification side memory $\mathcal{P}$ is a **safety property** precisely when

$$\rho_0 \in \textbf{Init}\ \wedge\ \rho_0 \rightarrow^* \rho\ \Longrightarrow\ \mathcal{P}(\rho)$$

where **Init** is the set of **valid initial states**. One way to achieve this is to add as an **Iris invariant reachinv**(ρ_0) that the current configuration of the PlusCal program is reachable from some initial configuration ρ_0. Now, assuming of course that $\rho_0 \in$ **Init**, we know that the current configuration satisfies $\mathcal{P}$. In the present case, a safety property that PlusCal/TLA+ can prove is:

$$\mathcal{P}(\rho)\ :=\ p \neq q\ \Longrightarrow\ \neg\rho(\texttt{pc}_p\texttt{=="cs"})\ \vee\ \neg\rho(\texttt{pc}_q\texttt{=="cs"})$$

stating that at any point in time, *at most one* thread can be in the critical section.

Getting the usual specification back In the usual specification of locks in concurrent separation logic, ever since the original CSL paper, and in particular in Iris, a lock *protects* some separation logic predicate $\mathcal{I}$ that we call the **invariant of the lock**. The specification we will deduce is the following:

$$\{\mathcal{I}\}\ \texttt{newlock()}\ \{l.\ \texttt{IsLock}(l, \mathcal{I}) * \circledast_{n=1}^{N} \texttt{Unlocked}_n(l)\}$$

$$\{\texttt{IsLock}(l, \mathcal{I}) * \texttt{Unlocked}_n(l)\}\ \texttt{lock(l)}\ \{\texttt{Locked}_n(l) * \mathcal{I}\}$$
$$\{\texttt{IsLock}(l, \mathcal{I}) * \texttt{Locked}_n(l) * \mathcal{I}\}\ \texttt{unlock(l)}\ \{\texttt{Unlocked}_n(l)\}$$

Metaremark 2. *It may be possible to add dynamically new instances of the lock, by replaying the whole simulation with more instances of the process, with the added processes idling. However, in the case of the FastMutex algorithm, one cannot escape this limitation of the number of threads using the lock, for it is consubstantial with the algorithm.*

We will deduce this higher-order specification from the simulation-based specifications above. To do so, we use the following Iris invariant:

$$(\exists n.\ \texttt{pc}_n \overset{\frac{1}{2}}{==} \texttt{"cs"})\ \vee\ \mathcal{I}.$$

Hence, the fact that some thread in the specification program is in the critical section becomes a token which one can exchange against the predicate $\mathcal{I}$ that is protected by the lock. Notice that this does not rely *at all* on the implementation of the lock on either side. Formally, we use the following definitions:

$$\texttt{Locked}_n(\gamma) := \boxed{\texttt{pc}_n \overset{\frac{1}{2}}{==} \texttt{"cs"}}^{\gamma}$$
$$\texttt{Unlocked}_n(\gamma) := \boxed{\texttt{pc}_n\texttt{=="ncs"}}^{\gamma}$$
$$\texttt{IsLock}(\gamma, l, \mathcal{I}) := \texttt{islock}(\gamma, l) * \boxed{(\exists n.\ \texttt{pc}_n \overset{\frac{1}{2}}{==} \texttt{"cs"})\ \vee\ \mathcal{I}}$$

The way the proof works is by using the following fact:

$$\forall p, q,\ (\texttt{pc}_p\texttt{=="cs"} * \texttt{pc}_q\texttt{=="cs"})\ \mathrel{-\!\!*}\ \bot\ .$$

(for the case $p = q$, we need to appeal to properties of the separation product). Then, we know that if we have a proof of $\texttt{pc}_n$=="cs" for some n, it must be the case that the Iris invariant for the lock contains the invariant $\mathcal{I}$, for it is impossible that the left disjunct be true.

3 Embedding PlusCal into Iris

One idiosyncrasy of PlusCal is that some statements are labelled (similarly to the labels goto uses in C), and execution between two labelled statements is *atomic*.

3.1 PlusCal configurations

A PlusCal configuration is a triple (*glob*, *tlocs*, *konts*). Both *tlocs* and *konts* are finite maps with integer domains. *glob* and each *tlocs*$[n]$, for each thread n, is a finite map from variable names to values, and *konts*$[n] = (stmt, k)$ is the pair of the next statement to execute and the continuation of that instruction in thread n. The natural thing to do would be to

use the authoritative monoid with it, and have the domain of the maps be exclusive. We define the following notations:

- $\mathsf{x}{==}v$ for $\circ([x := v], \emptyset, \emptyset)$,
- $\mathsf{x}_t{==}v$ for $\circ(\emptyset, [t := [x := v]], \emptyset)$,
- $\mathsf{pc}_t{==}lab$ for $\exists stmt, k, \circ(\emptyset, \emptyset, [t \mapsto (lab{:}stmt, k)])$

The idea is that if someone holds some piece of the specification heap, say $\mathsf{x}_t{==}v$, this constrains the transitions we can choose for the specification program in our simulation.

3.2 Move the simulation forward

Something like $\{P\}lab \rightsquigarrow lab'\{Q\}$? and then we can use it to change the state of the specification program? Maybe it's simply $(P * \mathsf{pc}_n{==}\texttt{"l"}) \Rrightarrow (Q * \mathsf{pc}_n{==}\texttt{"l'"})$?

4 Bigger example

The following example is less trivial than the first one. It taken essentially verbatim from "The PlusCal algorithm language" by Lamport. The corresponding HeapLang implementation should have basically the same specification in terms of labels as the spinlock example.

```
--algorithm FastMutex {
  variables x, y= 0, b = [i \in 1..N |-> FALSE];
  process (Proc \in 1..N)
    variable j;
  {
      ncs: while (TRUE) {
             skip; \* the noncritical section
    start:   b[self] := TRUE;
      s01:   x := self;

      s02:   if (y # 0) { s03: b[self] := FALSE;
                          s04: await y = 0;
                               goto start; };

      s05:   y := self;
      s06:   if (x # self) { s07: b[self] := FALSE;
                                  j := 1;
                           s08: while (j <= N) { await ~b[j];
                                                 j := j+1; };
                         s09: if (y # self) { s10: await y = 0;
                                                   goto start; };
             };

       cs:   skip \* critical section
      s11:   y := 0;
      s12:   b[self] := FALSE;
           } \* end outer while

  }
}
```

Figure 2. Fast Mutex