# Proving with models

## 1 Introduction

The general goal is to understand how one can incorporate models in order to prove programs using Hoare-style logic. The concrete instantiation of this idea that we propose is the following. Say our programmer wants to prove some program or some library; they would proceed like this:

1. they write their program in HeapLang;
2. they write an abstract version in some model checking tool, for example in PlusCal for the TLA+ system;
3. they prove in Iris that the two are in correspondence: the implementation refines the abstract version;
4. they use their verification tools to prove/gain confidence in some properties of their model;
5. they use these properties to prove Iris-style specifications of their HeapLang program.

I think this is interesting at the conceptual level, because it is a way to understand the relationship between two different styles of specifications. On the one hand, one can specify a lock as a transition system with two states and two transitions (apparently that is what TaDa is doing?), and on the other hand the Iris-style with an invariant. It would be interesting to understand what the relationship between these two styles are, and one can see that as one step in that direction.

**Note 1.** *Less related to this work, but in the same spirit of interaction between interactive and automatic provers: I believe that, if we are to "routinely" prove large programs, we need to be able to mix automated techniques and interactive proofs for the most subtle parts. On example of a work that is aiming at this hybrid proof methods is [2]. The idea is to prove the difficult part of the program, for example the parts that involve complicated algorithms and data structures, using an interactive tools; and prove the more mundane parts of the program, which use the aforementioned parts, using automated tools. This means that we would have to be able to formalize* what *the automated tools are doing to be able to prove that if they find the program correct while assuming the specifications that we proved in Iris, then the program is correct indeed.*

### 1.1 Models in Iris

The details are in the other document, I do not have the time to integrate this before the weekend.

The idea is that we put the abstract transition system (ASTS) in the Iris ghost state, and we prove Hoare triples where the current ASTS is in the pre- and post-condition. This amounts to saying that executing that piece of code will *move* the abstract state from the one in the precondition to the one in the postcondition. In the case of PlusCal, the state consists basically in the memory state of the program and of one program counter (pc) for each thread. Which means that we can put the actual current ASTS state in an invariant and then give to each HeapLang thread a non-authoritative ownership of one of the program counters that constitute the state of the ASTS.

Then, in the lock example, what TLA+ is giving us is that at most one thread has its pc equal to the pc of the critical section. In the Iris would, this means that ownership of that pc can be used as a token, recovering (almost) the usual Iris specification of a lock.

### 1.2 Deadlocks

To prove some kind of deadlock freedom, one would first make deadlocks an error in the semantics of the programming language, since all the soundness theorem can tell us is that the program will not crash, essentially. Then, we would just need to get an ASTS that represents the communication patterns of the protocol and a proof that we never attain a state that is deadlocked. Then, a simulation between the program and the ASTS should give directly the impossibility of deadlocks. For this we can pull the singularity OS out of the freezer, since their system should prevent (binary) deadlocks.

### 1.3 Unification

The goal is to be able to express both the PlusCal/TLA+ and the SingularityOS protocols logics by reducing them the same notion of STS that we already have developed for the termination work.

## 2 The base layer: STS and simulation

**Note 2.** *This corresponds to the setting of our termination work. I have chosen to keep, for now at least, the definitions of the weakest precondition that were meant to preserve termination.*

In this section, we present the general framework, where the whole logic is parameterized with a state transition system, which we call the *abstract state transition system* as it will live in the logical state.

**Definition 2.1** (ASTS)**.** An **abstract transition system (ASTS)** $\mathcal{A}$ is the data of: 1. a set $\mathcal{S}(\mathcal{A})$ of **states**, 2. a set $\mathcal{R}(\mathcal{A})$ of **roles**, 3. a set $\mathcal{T}(\mathcal{A})$ of **transitions** $t$, which each has a source and a target state, and a role $\lambda(t)$, 4. a distinguished initial state $s_0^{\mathcal{A}} \in \mathcal{S}(\mathcal{A})$.

The only non-standard notion here are roles: their purpose is to express the fact that only one specific thread of

Version as of Friday 22nd May, 2020 at 18:30.

the program is allowed to take each transition. When we instantiate the framework, each thread will be given at most a single role from each instance of an ASTS.

The reasoning principle that this will induce is that between two turns of some thread in the execution schedule, we know that no transition associated to its role has been taken, which gives us information about how the state of the ASTS has changed. This is achieved through the following two predicates:

$$\mathbf{Now}^{\mathcal{A}}(s) \qquad\qquad \mathbf{Last}^{\mathcal{A}}_{\rho}(s)$$

The first predicate, $\mathbf{Now}^{\mathcal{A}}(s)$, for a state $s \in \mathcal{S}(\mathcal{A})$ of the ASTS $\mathcal{A}$, simply asserts that the current state of that ASTS is $s$. The second predicate, $\mathbf{Last}^{\mathcal{A}}_{\rho}(s)$, for a role $\rho \in \mathcal{R}(\mathcal{A})$ and a state $s \in \mathcal{S}(\mathcal{A})$, represents the thread with role $\rho$'s knowledge of current state of the ASTS. This knowledge is not complete since other threads may have advanced the state of the ASTS since it was last scheduled. However, only a thread with role $\rho$ can update the state of the ASTS along a transition labeled with the role $\rho$. This means that, from $\mathbf{Last}^{\mathcal{A}}_{\rho}(s) * \mathbf{Now}^{\mathcal{A}}(s')$, we can deduce that $s'$ is reachable from $s$ though a path that *does not* contain any $\rho$-transition.

**Example 2.2.** Consider an ASTS whose states made up of one piece of state for each role, that is, such that:

$$\mathcal{S}(\mathcal{A}) = \mathbf{S}^{\mathcal{R}(\mathcal{A})}$$

for some set $\mathbf{S}$. Suppose moreover that transitions are only allowed to change the component of the state that corresponds to their role. Then the reasoning principle above implies that the frame preserves the portion of state that corresponds to the current thread, as is usual for separation logic. The case of PlusCal programs will be a generalization of that case, see Section 4.

### 2.1 State tracking with Iris resources and invariants

Let us see how these two predicates are implemented in terms of Iris resources and invariants. The idea is that, in addition to the current state of the ASTS, we need to remember the path we have taken to reach this path, in order to know which transitions were *not* taken. Our solution is to consider the bag (multiset) of the roles of the transitions that have been taken, instead of considering the paths themselves. This is needed to obtain a *commutative* monoid.

Formally, given a path $p$ from a state $s_0 \in \mathcal{S}(\mathcal{A})$ to a state $s_1 \in \mathcal{S}(\mathcal{A})$ (which we write $p : s_0 \twoheadrightarrow s_1$), define its **history** to be the bag of all the roles associated to the transitions of $p$. Given a history $h$ (a bag of roles of $\mathcal{A}$) and two states $s_0$ and $s_1$, we say that $s_0$ **is reachable from** $s_1$ **through** $h$ if there exists a path $p : s_0 \twoheadrightarrow s_1$ such that $h$ is the history that corresponds to $p$. We define the **difference** of two histories in the following way. Given three histories $h$, $h_1$ and $h_2$, we write $h = h_2 \setminus h_1$ if, when we consider histories as maps from $\mathcal{R}(\mathcal{A})$ to $\mathbb{N}$ with finite support, for all roles $\rho$, $h(\rho) = h_2(\rho) - h_1(\rho)$. We keep track of **checkpoints**, which are pairs $(h, s)$ of a history $h$ and a state $s \in \mathcal{S}(\mathcal{A})$. We use the following resources:

$$\begin{aligned}\gamma_{\text{hist}} &: Gmap(\mathcal{R}(\mathcal{A}), \mathbb{N})\\ \gamma_{\text{chkpts}} &: Auth(Gset(Product(Gmap(\mathcal{R}(\mathcal{A}), \mathbb{N}), \mathcal{R}(\mathcal{A}))))\\ \gamma_{\text{state}} &: AuthFrag(\mathcal{S}(\mathcal{A}))\end{aligned}$$

The resource named $\gamma_{\text{hist}}$ contains the current history, which gives information about which transitions have been taken to arrive at the current state, contained in $\gamma_{\text{state}}$. The three resources are related by the following invariant:

$$\exists s, h, H, \boxed{\circ\,(1/2, s)}^{\gamma_{\text{state}}} * \boxed{\bullet\, h}^{\gamma_{\text{hist}}} * \boxed{\bullet\, H}^{\gamma_{\text{chkpts}}} * \\ \forall (s_i, h_i) \in H, s \text{ is reachable from } s_i \text{ through } h \setminus h_i$$

In particular, assuming $(\emptyset, s_0^{\mathcal{A}}) \in H$, this invariant implies that the current state of the ASTS is reachable from the initial state $s_0^{\mathcal{A}}$.

The definition of $\mathbf{Last}^{\mathcal{A}}_{\rho}(s)$ leverages this invariant by asserting the existence of a checkpoint whose history is related to the *current* history. More precisely, it asserts that the current history contains the same number of $\rho$-transitions as the history in the checkpoint: the predicate is thus defined as

$$\exists n, h, \boxed{\circ\,[\rho \mapsto n]}^{\gamma_{\text{hist}}} * \ulcorner h(\rho) = n \urcorner * \boxed{\circ\,\{(h, s)\}}^{\gamma_{\text{chkpts}}}$$

This means, together with the invariant above, that the current state $s'$ is reachable from $s$ through $h \setminus \tilde{h}$, for some history $\tilde{h}$ whose component $\tilde{h}(\rho)$ at $\rho$ is equal to $h(\rho)$, since the predicate $\mathbf{Last}^{\mathcal{A}}_{\rho}(s)$ owns the fragment of the current history corresponding to $\rho$. Hence, we know that the current state $s'$ is reachable from $s$ without going through a $\rho$-transition.

Finally, the predicate $\mathbf{Now}^{\mathcal{A}}(s)$ is simply an alias for the second half of the current state of the abstract state transition system:

$$\mathbf{Now}^{\mathcal{A}}(s) \quad = \quad \boxed{\circ\,(1/2, s)}^{\gamma_{\text{state}}}$$

### 2.2 Inference rules

All the rules we present here assume the presence of the invariant that we described in the previous section. In practice, since the current state of the ASTS is shared between the different threads of the HeapLang program, it is put inside an invariant that is custom to the user of this ASTS library, as we will see when we describe the case studies in Sections 3 and 4.

As we explained earlier, we can deduce information about the current state from the thread local view of the current state of the ASTS:

$$\mathbf{Last}^{\mathcal{A}}_{\rho}(s) * \mathbf{Now}^{\mathcal{A}}(s') \;\Rrightarrow \\ \mathbf{Last}^{\mathcal{A}}_{\rho}(s') * \mathbf{Now}^{\mathcal{A}}(s') * \ulcorner \exists p : s \twoheadrightarrow s' \wedge \rho \notin p \urcorner$$

where we write $\rho \in p$ when the path $p$ contains a transition whose role is $\rho$. This rule also updates the last known state of the ASTS from the point of view of the thread $\rho$.

Naturally, it is also necessary to be able to update the current state of the ASTS. This is achieved in the following way:

$$\mathbf{Last}_\rho^{\mathcal{A}}(s) * \mathbf{Now}^{\mathcal{A}}(s) * \ulcorner \exists p : s \overset{\rho^*}{\twoheadrightarrow} s' \urcorner \;\Rrightarrow\; \mathbf{Last}_\rho^{\mathcal{A}}(s') * \mathbf{Now}^{\mathcal{A}}(s')$$

In words, if the current state of the ASTS is $s$, thread $\rho$ is able to able to update it to any state $s'$ which is reachable from $s$ though $\rho$-transition.

### 2.3 Weak adequacy

The invariant associated to the ASTS $\mathcal{A}$ implies in particular that for each checkpoint $(h, s)$ of a history and a state of the ASTS which is a member of the set $\gamma_{\text{chkpts}}$, the current state of the ASTS is reachable from $s$. In practice, the initial checkpoint $(\emptyset, h)$ will always be in that set, which means that the current state is always reachable.

The reason this is useful is that it lets us leverage *safety properties* on the abstract state transition system. Indeed, a safety property states that any state which is reachable from the initial state is not a *bad state*, for some fixed state of bad state. Since we are interested in proving safety properties of our HeapLang programs, this simple reachability property is what we need.

### 2.4 Stronger adequacy and sound*i*ness

The adequacy which we stated in the previous section only needs the usual Iris weakest precondition. We can instead use a stronger weakest precondition which talks directly of the roles of the threads and of the state of the abstract state transition system. This new weakest precondition would be guarantee that each physical thread corresponds indeed to at most a single role in the ASTS. However, this is not quite clear how we would/could use this information to prove safety properties of our programs.

Moreover, this other weakest precondition *seems to guarantee* that each thread is productive, in that it take only finitely many steps before performing a transition in the abstract state transition system. This would imply, for instance, that is the ASTS only contains finite paths, the programs would be terminating. It is, however, unclear that this is provable in the current version of Iris.

## 3 Instance: Singularity's protocols

Singularity OS was an experimental operating system, based on the idea that one could favorably replace the virtual memory protection and isolation facilities of CPUs by a managed runtime executing well-typed programs[1]. Here, we focus one one small aspect of this project: the message passing mechanism. In the same philosophy of replacing dynamic checks with static guarantees on untrusted but well-typed programs, they devised a very efficient implemenation of interprocedural message passing.

In more details, each communicating pair of channels are typed according to a *protocol*, which describes which messages can be sent or received at each step of the protocol. Then, a specialized compiler analyses the protocol and generates code for the send and receive operations for each step. One of the things that this analysis does it to bound the size of each process' mailbox, which means that there is no need for dynamic allocation, or even bound checking.

In this section, we explain how a protocol induces an ASTS which we can use to derive Iris Hoare triples for the send and receive operations. The interesting part is that, though the soundness relies on the static bound which the process analyser gives us, the Hoare triples allow for the usual higher order specifications.

### 3.1 Singularity protocols

A Singularity OS protocol is composed of a set of states, wich a distinguished initial state, and, for each state, the messages can be sent or received, and to which state it moves the protocol to. Such a protocol is described using a simple domain-specific language as in Figure 1. As the protocols are used to decribe interactions between a client and a server, it is sufficient to describe it from the point of view of, say, the server, and the protocol of the client is deduced by duality.

In this example, at the begining of the protocol, the client must receive –as indicated by !, as opposed to ? for reception– a DeviceInfo message to the server. The protocol is then in the state IO_CONFIGURE_BEGIN, where the client must send two messages in a row to register and set parameters. It then must wait for acknowlegments from the server: either the client received a InvalidParameters message, in which case we start again from IO_CONFIGURE_BEGIN, of we succeed and to the IO_CONFIGURED state. As demonstrated by this last step, the flow is determined by the kinds of the messages which are received, and the two peers are assured to stay synchronized as long as they both follow the protocol because message delivery is reliable. Note that they are synchronized up to a delay between sending and reception of the messages.

More formally, a protocol is a state transition system with the same states as the protocol and a transition labelled with the kind and the direction of the message between states.

### 3.2 Embedding into Iris

The goal is to deduce from the protocol Hoare triples which mimic a linear type system of the channels, where the type of a channel changes when messages are sent or received. For example, the implementation of the server of the protocol of Figure 1 would be using Hoare triples such as:

[1] Those were more optimistic times, before Spectre.

```
contract NicDevice {
  out message DeviceInfo (...);
  in message RegisterForEvents(NicEvents.Exp:READY evchan);
  in message ConfigureIO();
  in message PacketForReceive(byte[] in ExHeap pkt);
  out message BadPacketSize(byte[] in ExHeap pkt, int mtu);

  state START: one {
    DeviceInfo ! -> IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents ? -> SetParameters?
                  -> IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters ! -> IO_CONFIGURE_BEGIN;
    Success ! -> IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO ? -> IO_RUNNING;
    ConfigureIO? -> IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? -> (Success ! or BadPacketSize!)
                  -> IO_RUNNING;
    GetReceivedPacket? -> (ReceivedPacket! or NoPacket!)
                  -> IO_RUNNING;
  }
}
```

**Figure 1.** Protocol from Singularity OS

```
{ c@START } c.SendDeviceInfo(v) { c@IO_CONF_BEGIN }
{ c@IO_CONFIGURED } c.Recv() { m.
                        match m with
                        | StartIO => c@IO_RUNNING
                        | ConfigureIO => c@IO_CONFIG_BEGIN
                        end}
```

### 3.3 Formalization

We do not instanciate the framework with the protocol seen as an STS, as the real STS which describes the executions of the protocol need to take into acount the points of view of both the server and the client. To that purpose, we define **the ASTS of runs of a protocol**. Its states are quadruples $(s_s, s_c, m_s, m_c)$ of the state of the server, the state of the client, the mailbox of the client, the mailbox of the server. The mailbox is the queue of messages that the other partcipant has sent but which have not been received yet. In this ASTS, when a message is sent, it is appended to the other's mailbox; when the mailbox is empty, the system follows a loop transition, not changing the state, otherwise the first earliest message of the mailbox is removed. There are two roles, one for the server and one for the client.

The Iris predicate c@STATE is defined as the existence of a state $s$ of $\mathbf{Runs}(P)$ whose first or second component (depending on whether c is a server or client channel) is STATE and such that $\mathbf{Last}_\rho(s)$ holds.

### 3.4 Implementation of message passing

Let us now focus on how message passing is implemented. Each pair of channels is a pair of fixed length arrays, used as a ring buffer. The length of the buffer is determined by an automatic analysis of the protocol which computes a bound on the possible size of the mailboxes. It rejects the protocol if no such bound exist, which corresponds to the case where one end can send messages forever without needing to receive; in other words, if there is a cycle in the protocol with only sends or receives.

Because of this static guarantee, the sender does not need to check for an overlfow before appending its message. Thanks to the bound on the size of the mailboxes in the ASTS $\mathbf{Runs}(P)$, we can prove that such a program is indeed safe.

### 3.5 Proving the absence of some deadlocks

**Todo 1.** *By instrumenting the semantics, we can also rule out two threads deadlocking by both waiting to receive on both endpoints of the same channel when both mailboxes are empty.*

## 4 Instance: PlusCal programs as resources

In this second case study, we show how to use TLA+ specification and its accompanying model checker to prove modules in the concurrent separation logic. TLA+ is a tool to specify systems developed by Lamport and his collaborators. The user specifies their system using a variant of set theory augmented with constructions from temporal logic. When the system in question is more easily expressed as an algorithm than a temporal logic formula, one can use the PlusCal language [1] which is a simple imperative language which is translated into a TLA+ formula. This language is not meant to be a programming language however, as it supports operations such as set comprehensions and a Hilbert choice operator. The reason for this design is to be able to write programs in the style of pseudo-code, where programming constructs are interleaved with more logical statements.

We demonstrate how to relate a module written in the HeapLang language to a PlusCal program, and to deduce the correctness of the specification of the HeapLang program from a safety property of the PlusCal program. Our first example is a locking module, first we consider a simple spin lock implementation, and then a more involved one. One interesting aspect of this example is that we use idiomatic specifications for both sides. For the Iris side, a lock is specified are a way to protect an invariant, any separation logic predicate; this invariant becomes available to the program when the lock is unlocked, and must be given back to the lock when releasing the lock. For the TLA+ side, the whole system is represented as a single program running identical code on a certain number of processes; see Figure 2 for the implementation of a spinlock in PlusCal and in HeapLang.

Surprisingly, the PlusCal implementation is not composed of several function, instead it is a single program which

represents all the allowed interaction patterns which are allowed with a lock: there are $N$ threads each locking and unlocking the same lock infinitely many times. What would be the arbitrary client code around the calls to acquire and release in HeapLang are represented as two skips in the PlusCal program. This is reasonable because, as long as the client code does not touch the memory that is private to the implementation of the lock, we do not care about what it does. This kind of reasoning is at the core of separation logic and its Frame rule.

Thus, in deriving the usual specification for locks in Iris from the PlusCal one, we justify and make explicit the assumptions on which the PlusCal specification relies.

### 4.1 PlusCal and TLA

The semantics of PlusCal programs is a little unusual, we point out where diverges from usual imperative languages here. First, the atomic unit of execution is not the statement. Instead, the program executes atomically between two labels; in Figure 2, the two labels are ncs and cs. There are restrictions about where labels must be added with respect to control flow constructions, see the PlusCal documentation for more details. Second, the construction await P waits until the property P becomes true. Finally, the program can stutter, in that, for each state, there is a transition in its semantics to itself. This is how PlusCal deals with the fact that a program lives in an unknown environment.

The TLA+ toolchain translates such a PlusCal program into a TLA+ formula of the form $Init \wedge \Diamond Step$, where $Step$ is a formula relating the values of the programs at one time to their values at the next step. It keeps track of where in the execution of each process it is using an auxiliary variable pc containing a label.

This variable is important to specify some programs: The specification of the spin lock says that it is always the case that:

$$\forall p, q, \mathrm{pc}[p] = \mathrm{pc}[q] = \text{ cs } \text{''} \implies p = q.$$

In other words, two different threads *cannot* both be in the critical section at the same time, which is the direct definition of a critical section.

```
--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
          } } }
```

**(a)** PlusCal implementation

```
let newlock () = ref false

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

let release l = l := false
```

**(b)** Heaplang implementation

**Figure 2.** Spinlock implementations

### 4.2 Semantics of PlusCal programs

Unlike its definition in TLA+ as a translation into a TLA+ formula, we give a usual small-step sematics to PlusCal programs.

The grammar of the language is the following. The statements are:

$$\begin{aligned} C ::= {} & x := E \mid \mathsf{if}\ (E)\ \{C_1\}\ \mathsf{else}\ \{C_2\} \mid \mathsf{either}\ C_1\ \mathsf{or}\ C_2 \\ & \mid \mathsf{while}\ (E)\ \{C\} \mid \mathsf{with}\ (x \in E)\{C\} \mid \mathsf{await}\ E \\ & \mid \mathsf{assert}\ E \mid \mathsf{skip} \mid \mathsf{return} \mid \mathsf{goto}\ \ell \mid \ell : C \end{aligned}$$

where $\ell$ is a program label, and $E$ is an expression.

The semantics which we use here is similar to the semantics of Clight in Compcert[**compcert**].

First, the notion of program state that we use contains a statement and a continuation. Intuitively, the sequence of execution is, first, to execute the statement, and then to execute the continuation. One of the reason we need to use explicit continuations is that the PlusCal language contains the goto statement, which manipulates this continuation. This plays roughly the role of executions contexts in the semantics of HeapLang, except that here we use it explicitely. The continuations are defined inductively by:

$$k ::= \mathsf{stop} \mid C\,;k$$

Let us write $\langle C \mid k \rangle$ for the pair of a statement $C$ and a continuation $k$. Then, the state of the program is given by

$$(\langle C \mid k \rangle, \mathfrak{s}_g, \mathfrak{s}_\ell)$$

where $\mathfrak{s}_g$ is the global state of the program, and $\mathfrak{s}_\ell$ is the thread local state of the program, indexed by thread identifier.

We give an exerpt from the pre-semantics: the part which deals with control flow in order to demonstrate how the continuations are used. For this reason, we omit the part of the state that has to do with memory.

$$\begin{aligned} \langle C_1; C_2 \mid k \rangle &\to_\epsilon \langle C_1 \mid C_2\,; k \rangle \\ \langle \mathsf{while}\ (\mathsf{E})\ \{C\} \mid k \rangle &\to_\epsilon \langle \mathsf{while}\ (\mathsf{E})\ \{C\} \mid k\,; \rangle \\ \langle \mathsf{goto}\ \ell \mid k \rangle &\to_\epsilon \langle C' \mid k' \rangle \end{aligned}$$

where, in the second rule we assume that the condition $E$ is true, and in the third and last rule $\mathsf{findLabel}(\ell) = (C', k')$. The (partial) function findLabel finds a label in the body of

the program and builds a pair of a statement and a continuation which represents the execution of the program starting from that label.

Because execution is atomic between two labeled statements, we define the semantics of the language as a sequence of steps in the pre-semantics with:

$$\frac{p : s_1 \rightarrow_\epsilon^* s_n \text{ where } s_1 \text{ and } s_2 \text{ are the only labeled states in } p}{s_1 \rightarrow s_n}$$

### 4.3 Instanciation in Iris

We choose to give directly an operational semantics to PlusCal programs instead of going through their translation to TLA+ formulas. In future work, we could prove adequacy of that semantics with respect to the semantics of TLA+, but this would be a endeavor to itself.

The semantics of PlusCal programs is is given as an abstract state transition system. Its set of states is:

$$(Vars \rightharpoonup_{fin} Val) \times (\mathbb{N} \rightharpoonup_{fin} (Vars \uplus \{\mathrm{pc}\} \rightharpoonup_{fin} Val)).$$

The first component represents the memory which is shared between the processes, represented as a finite partial map from variables to values. The second represents the local state: We associate a memory state to each process identifier which exists. For simplicity, we consider pc not to be a valid identifier, and we associate a value to it for each process. Given a variable $x$ a process identifer $p$ and a state $\mathfrak{s}$, we write $\mathfrak{s}[x]$ for the value of the shared variable $x$, and $\mathfrak{s}_p[x]$ for the value of the local variable $x$ of process $p$.

The roles are simply defined to be the process identifiers (natural numbers, that is), and transitions correspond to atomic steps of the program, annotated with the identifier of the process which performed it. Of course, a simple invariant of the semantics is that only a process can access its local memory, which means that we are in a situation similar to Example 2.2.

In practice, we will often relate on thread in the HeapLang side to one process in the PlusCal side. We essentially treat the current state of the PlusCal program in the same way as the HeapLang programs physical memory: We put it in an invariant like the following to relate it with a more refined resource

$$\exists \mathfrak{s}, \mathbf{Last}_p^{\mathcal{A}}(\mathfrak{s}) * \boxed{\bullet\, \mathsf{inj}(\mathfrak{s})}$$

where inj embeds the physical state into the resource we use to reason about its ownership. Concretely, we use the same RA as for HeapLang with fractional permissions, adapted to the non-flat memory of PlusCal. We write $x \xrightarrow[i]{p} v$ for the ownership of a fraction $p$ of the variable $x$ of thread $i$, and $x \xrightarrow{p} v$ when $x$ is a shared variable. As usual, we omit the permission $p$ when it is equal to 1.

### 4.4 A program logic for PlusCal ghost programs

The framework of abstract state transition system gives us a reasoning principle stating that if we own the local control state of a thread and the memory, we get a viewshift to the state of the PlusCal program after any possible transition. However, from a practical point of view, this is not practical because each transition of the PlusCal program is a *sequence* of what would be a transition in a usual programming language.

This is why we define a predicate $\mathrm{step}_i p_f \{Q\}$ which states that we can update the logical state to a world where the local control state is $p_f$ and the predicate $Q$ holds. The definition is similar to that of the weakest precondition, specialized to the idiosyncrasies of PlusCal and to the fact that the program state is in the ghost state.

### 4.5 Example: mutual exclusion

As a first example, we derive the usual Iris specification of locks from the property that at most one process of the PlusCal program in Figure 2 can be at the label cs at the same time. Formally, the TLA+ model checker assures us of the fact that:

$$\forall \mathfrak{s}, \mathfrak{s}_0 \rightarrow^* \mathfrak{s} \Rightarrow \forall p, q, \mathrm{pc}_{\mathfrak{s}}[p] = \mathrm{pc}_{\mathfrak{s}}[q] = \texttt{"cs"} \implies p = q.$$

The basic idea is that when the program allocates a new lock with newlock, in the logical side, we allocate a new instance of the PlusCal program with $N$ threads. In this section, we assume that the lock will be used in a fixed number $N$ of thread, we will explain how to deal with dynamic creation of PlusCal threads later.

We give the following specification to the lock module in Figure 2:

$$\{\mathcal{I}\}\ \texttt{newlock ()}\ \{l.\ \exists \gamma, \mathtt{islock}(\gamma, l, \mathcal{I}) * \circledast_{n=1}^{N} \mathtt{lockable}(\gamma, n)\}$$

$$\{\mathtt{islock}(\gamma, l, \mathcal{I}) * \mathtt{lockable}(\gamma, n)\}\quad \texttt{acquire l}\quad \{().\ \mathtt{locked}(\gamma, n) * \mathcal{I}\}$$
$$\{\mathtt{islock}(\gamma, l, \mathcal{I}) * \mathtt{locked}(\gamma, n) * \mathcal{I}\}\quad \texttt{release l}\quad \{().\ \mathtt{lockable}(\gamma, n)\}$$

where we define

$$\mathtt{lockable}(\gamma, n) \coloneqq \boxed{\mathrm{pc}_n \rightarrow \texttt{"ncs"}}^{\gamma}$$
$$\mathtt{locked}(\gamma, n) \coloneqq \boxed{\mathrm{pc}_n \xrightarrow{1/2} \texttt{"cs"}}^{\gamma}$$
$$\mathtt{islock}(\gamma, l) \coloneqq \boxed{(\exists n.\ \mathrm{pc}_n \xrightarrow{1/2} \texttt{"cs"}) \ \vee\ \mathcal{I}} * \boxed{\exists v.\ l \mapsto v * \boxed{\mathtt{b} \rightarrow v}^{\gamma_{\mathrm{PC}}}}$$

Here, $\gamma_{\mathrm{PC}}$ is the identifier for the resource which contains the abstraction of the PlusCal heap, provided by the invariant relating the state of the PlusCal ASTS and the abstract heap. The token which is held by the thread which owns the lock is the the fact that its correponding thread in the abstract program is in the critical section, as witnessed by its associated pc variable. The safety property which is provided by the TLA model checker is then used to prove that at most one thread can hold this token.

This suggests that the style of specification used by TaDa and Hur (??), which models a lock with a two state state

transition system can be used to prove the CSL style of specification which is used in Iris. We believe that this warrants a more formal investigation.

### 4.6 Dynamic creation of PlusCal threads

Basically, when we add a new thread, we map the path from the $N$-threaded initial state to the current $N$-threaded state to a $(N + 1)$-threaded path.

## 5 Related works

### 5.1 Wytse Oortwijn, Marieke Huisman*, et al*

What they do is most similar to what I was proposing, where there were abstract programs in the ghost state, which were similar to Kleene algebras with tests and parallel product. This means that the abstract program and the program they prove are basically the same. Our second example, with the TLA+ system, is most similar. However, we are interested in contrasting the two specification styles, both of the properties we wish our programs to satisfy, and how we write the program themselves. In a way, by deducing the Iris lock specification from the TLA+ one, we justify that the PlusCal lock program is a good specification for a lock module.

They seem to annotate the program itself to specify which abstract program each piece of code is meant to implement. I need to look at the soundness part to figure out whether they need to reflect this into the semantics, or if it is just a notational artifact.

## References

[1] Leslie Lamport. “The PlusCal Algorithm Language”. In: *Theoretical Aspects of Computing - ICTAC 2009*. Ed. by Martin Leucker and Carroll Morgan. 2009.

[2] Arseniy Zaostrovnykh et al. “A Formally Verified NAT”. In: SIGCOMM ’17. Los Angeles, CA, USA, 2017, pp. 141–154. ISBN: 9781450346535.