Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Compositional Reasoning about Concurrent Programs: TaDA 2.0

Supervisor:
Prof. Philippa Gardner

Author:
Ines Wright

Co-supervisor:
Dr. Daniele Nantes-Sobrinho

Second Marker:
Prof. Sophia Drossopoulou

June 19, 2024

Abstract

Despite much work on concurrent software verification in the last few decades, its practicality for use in large-scale software systems remains limited. In part, this is inherent in the balance between increasing the expressivity of the logics and reducing the complexity of the model the logic is checked against, meaning that logics with the power to verify the kinds of highly complex interactions we see in programs today frequently become complicated and therefore difficult to extend.

The introduction of TaDA, and later TaDALive, brought much increased expressivity to existing logics by allowing the verification of partial correctness properties of finer-grained code, and later total correctness properties. This thesis explores what it means to bring TaDALive's expressivity, generality and precision to TaDA, with a new concurrent separation logic: TaDA 2.0.

TaDA 2.0 provides a new semantic model based on that of TaDALive, in the process resolving soundness issues with both earlier logics, and then simplifies much of the semantic model and proof rules. Much of the technical work necessary for the soundness proof omitted in the earlier works is made explicit, as well as the details of verified example code, and it is concluded from these examples that TaDA 2.0 provides the same expressivity and modularity as TaDALive. Furthermore, this thesis conjectures that the simplifications to the logic are substantial enough for TaDA 2.0 to be extensible to more complex concurrency patterns such as helping.

Acknowledgements

I am, of course, indebted to my supervisor, Prof. Philippa Gardner, for her insights and guidance on this project, and for entrusting it to me. I am also grateful for her belief in me, as being able to trust it over my own I am certain inspired me to achieve far more than I otherwise would have, and for the opportunity to attend POPL'24, which was transformative to how I think about research. Likewise, Dr. Daniele Nantes-Sobrinho, for all the time she spent reading a latest set of definitions and proofs, preventing me from running away with unworkably complicated versions, and being a LaTeX whizz. Finally, Huan Sun for the time he spent finding the problems in early versions, his ideas towards simplicity, and for checking my example so last minute. Together, our discussions never failed to find new and exciting things to think about, whether that be a new solution to an existing problem or a new problem in an existing solution. It has been a pleasure working on this with you all.

I would be remiss in not thanking Prof. Kevin Buzzard for his supervision in the earlier years of my degree, without which I may never have had the mathematical maturity, attention to detail and independence required to succeed with a project like this, and Dr. Steffen Van Bakel, whose teaching found for me in PL the rigour and structure I was looking for.

Last but never least, my parents, who have supported me for the last four years and trusted me to make the right choices for myself, and my grandfather, who died before I followed in his footsteps, but I think would have been proud.

Contents

Chapter 1

Introduction

Software verification has risen in popularity in recent decades, as the complexity of the software we use on a day-to-day basis increases. Although useful for many things, it is clear that existing testing methods to ensure correctness of software systems are insufficient when the scale of software we use on a day-to-day basis is now large enough that it is impossible to ensure every component is bug-free, and more essential than ever as we increasingly rely on our software systems for potentially life-saving and life-endangering activities. Simultaneously, as the complexity of our programs increase, the need for ever-increasing speed from our processors fails to be met, and programmers reach for parallelism as an alternative, leading to nondeterminism in essential machinery which ca not easily to guaranteed to be correct with standard testing methods - a nondeterministic bug which causes total failure almost never will still eventually occur when software is used enough, and the consequences of these failures can be catastrophic. As a result, much research in software verification has been done to advance the field to handle the increasing complexities of the software we aim to verify, with tractable concurrent software verification for large-scale software still very much an open problem.

The increasing scale of software systems requires programmers to write their code in a *composable* way: it is essential for the clarity of the programmer to be able to write code in self-contained components which are easier to reason about at their natural level of abstraction, with software systems as a whole composed of the complex interactions of their many components. Similarly, a tractable software verification system requires composability: the user simply cannot think at once about every single component, and relies on being able to verify each software component separately and conclude a proof of the larger systems by composing these smaller proofs. Indeed, advanced mathematics is also approached in this manner, as an algebraic geometer simply can not work down to every definition, and instead applies lemmas of more abstract properties of each components.

It is clear that composability relates in some sense to *abstraction*: it is sensible to define and prove properties at different levels of abstraction. For example, verifying properties of a data structure such as membership or ordering do not necessarily have to be dependent on the underlying implementation, and indeed CAP [1] shows that it is necessary to express properties at the right level of abstraction in order to be able to compose components with the same behaviour and different underlying implementations or data representations.

Analogous to composability, is the need for *modularity*. While composability allows us to reduce each proof to a tractable size, modularity allows us to reuse proofs about our components in different contexts, so that we do not need to check the same property half a dozen times. Again, this corresponds to the way a programmer often thinks about their code: no one would write a new implementation of a lock for every data structure! They would simply use reuse the same one.

Work towards these goals has resulted in a long line of concurrent separation logics, as seen in Figure 1.1, building on each other in different ways to provide different strengths and weaknesses.

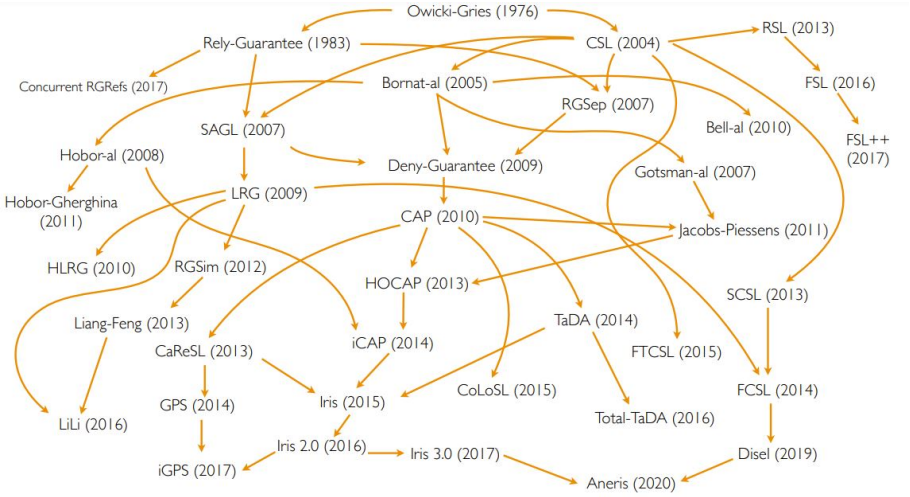

Figure 1.1: A variety of program logics for different types of concurrency [2]

In order for verification to really be considered practical for the scale and complexity of software systems in use in industry today, it is crucial that they are sufficiently *expressive*. It's increasingly common for large amounts of mutable data in software systems to be manipulated in a fine-grained manner, where the programmer relies on very low-level control of the underlying representation in order to guarantee correctness and deadlock-freedom. Much progress has been made on the correctness front in recent decades with the extensions of concurrent separation logics which control interference on shared memory very precisely. This is considered ownership-based reasoning. However, there are synchronization techniques for fine-grained data structures which depend not on ownership-based reasoning, but on the idea of *atomicity*: that if an action cannot be made to go wrong by interference and sequentially correct, then it cannot go wrong. An alternative line of research into linearisability and the associated atomicity based proofs has been fairly successful.

More recently, the two approaches were combined in TaDA [3], which has the expressivity required to reason about increasingly complex algorithms which make use of both ownership-based reasoning and linearisability properties within the same logic.

One measure of whether sufficient expressivity and simplicity has been achieved is the development of computer tools for software verification. A number of academic tools such as Gillian [4] and industry tools like [5] have arisen with success in verifying varying levels of safety and partial correctness properties.

There are a number of more complex concurrency patterns which occur naturally in software systems which TaDA is unable to verify. Unfortunately, the nature of its semantic model is such that the author expects that a complete reformulation is required to be able to verify patterns such as *helping* or *synchronization*. Meanwhile, research on verification of liveness properties has led to the introduction of TaDALive [6], including a novel trace-based semantic model with precise control over the interactions of environment and local threads, but the scale and complexity of which makes it difficult to extend to new concurrency patterns.

With all of these concerns in mind, I aim to provide a precise and sound formulation of a program logic for ownership-based and atomicity based proofs, with a new semantic model inspired by that of TaDALive, in such a way which is extensible for further work on new concurrency patterns. I call this logic TaDA 2.0.

1.1 Report Structure

Chapter 2 reviews the key contributions to the field of concurrent software verification necessary to understand the scope of this thesis. It assumes the reader is familiar with Hoare logic and traditional separation logic.

Chapter 3 provides an overview of TaDA [3] and TaDALive [6], with more detailed discussion on the original TaDA program logic and TaDALive's semantic model, aiming to provide the reader with some intuition on how TaDA 2.0 is designed to be used, and how the components of the semantic model will fit together. It assumes the reader is familiar with basic programming language semantics.

Chapter 4 and 5 present TaDA 2.0 in full. Chapter 4 defines in full TaDA 2.0's syntax and semantic model. Chapter 5 defines TaDA 2.0's specification language and syntactic judgement (a full set of proof rules with accompanying side conditions). In both, attention is drawn to the introduction or correction of components which were missing or imprecise in the original TaDA, and components which I have been able to simplify or correct from TaDALive.

Chapter 6 proves in full the soundness of a selection of proof rules, and covers the technical work required to reduce the soundness proof to a tractable statement. The soundness of several additional rules is in Appendix D.

Chapter 7 verifies a specification for a spin lock implementation in TaDA 2.0 which is analogous to that proven in TaDALive. The TaDA implementation fails to satisfy missing but necessary premises of certain proof rules, and therefore is not correct.

Chapter 8 details the impact and significance of the choices made in TaDA 2.0 compared to TaDA and TaDALive. Chapter 9 includes concrete recommendations for how to resolve solve of these limitations and avenues for further work opened up by TaDA 2.0.

1.2 Contributions

This project aims to address the need for a precise and sound formulation of a program logic with the capabilities of both ownership and linearisation based safety proofs in such a way which maintains the modularity and composability properties of existing logics and is genuinely extensible to more complex concurrency patterns. Its contributions are:

- A complete set of sound proof rules sufficient for safety proofs.
- A new trace-based semantic model for TaDA, beginning with stripping out the technical machinery required for liveness reasoning in TaDALive. I then make a number of significant changes which greatly simplify the soundness result, as well as make efforts to detail the properties which the model is designed to satisfy in order to be correct.
- A complete set of technical infrastructure required for the components of the soundness proof to fit together, which is not included in the existing literature. Also, explicit soundness proofs of many types of proof rules, including the rules for which proofs exist in TaDALive, proof rules which are new in TaDA 2.0, and a selection of others.
- Verification of a spin lock implementation. TaDALive demonstrates that this specification is strong enough to be used in a variety of fine-grained client code.

Chapter 2

Preliminaries

2.1 Separation Logic

Traditionally, Hoare logic has been used to verify pre- and postconditions for code, written

$$\{P\}\ C\ \{Q\}$$

The usual semantics of this notation would be: "If we begin the program C in a state with satisfies predicate P, then we will not crash (fault), and if we terminate, we terminate in a state which satisfies predicate Q". Observe that we do not make any guarantees about termination, as this would amount to solving the halting problem, but we do guarantee that non-terminating programs never crash. Hoare logic gives us the formalism we need to guarantee correctness, but for some time, preliminary work on software verification based on Hoare logic had been intractable for larger programs: alongside the need for generality of program logics to verify a wide range of programs, in order for a program logic to be useful, it must provide some aspect of modularity, i.e. we would like to verify different components of software separately, and 'stick' the proofs together, analogous to the modularity we expect from our software. However, existing logics did not scale well to large mutable data structures, as this requires proofs of disjointness of our data structures, resulting in exponential explosion in the size of our predicates [7]. Key to a wealth of new research in the field of software verification then was the introduction of separation logic by Reynolds et al. in [7]. Separation logic is based on earlier logics of bunched implications, with the key insight that if disjointness was encoded into the semantics of predicates, then we would not suffer such an exponential explosion in predicate size and our proofs would scale better to larger mutable structures. For predicates P and Q, the semantics of the separating conjunction $P \star Q$ refer to objects (usually concrete resources such as a heap) which can be divided into two *disjoint* components, one satisfying P and the other satisfying Q. Crucially, this allows us to describe several mutable data structures while guaranteeing 'separation' between them, and proofs of programs affecting one data structure can be soundly reused in contexts with more (but separate) structures.

2.2 Concurrent Separation Logic

In order to correctly verify safety properties of concurrent programs, our program logics must be able to ensure that *every* interleaving of threads is safe. Furthermore, to sensibly verify a postcondition, it must be the case that the result of the program is not dependent on the interleaving of threads - we say the code is *race-free*.
An intuitive but restrictive condition for concurrent programs to be race-free is the idea of disjoint concurrency - that two programs execute in parallel, but do not have any shared resources, and therefore cannot interfere with each other. Given that traditional separation logic's key insight is the disjointness of two resources connected by a separating conjunction, a restricted set of concurrent programs can be verified by the introduction of the following rule, in combination with the condition that C_1 does not modify variables free in P_2, C_2 or Q_2, and vice versa [8].

$$\frac{\{P_1\}C_1\{Q_1\} \qquad \{P_2\}C_2\{Q_2\}}{\{P_1 \star P_2\}\ C_1||C_2\ \{Q_1 \star Q_2\}}$$

A formal method for verification of programs with a limited amount of well-defined interference was presented in [9], and is commonly referred to as the Owicki-Gries method, but this was subsumed by more recent work of O'Hearn, Reynolds and Brookes in [8] and [10]. The newer work allows a more varied nature of interference, and constructs programs in such a way that disallowed forms of interference are implicitly forbidden by the proof rules, as opposed to requiring an additional check that the execution of C_2 cannot interfere in the proof of C_1 and vice versa, as in the Owicki-Gries method. This ultimately produces more modular proofs and therefore is more scalable. For these reasons, I present here the basic ideas used in O'Hearn's work and readers wishing to find a full semantic model and soundness proof should refer to Brookes' [10].

O'Hearn [8] extends the grammar of the commands of a typical while language with a construct to execute two commands in parallel, and a command for accessing a resource in a *conditional critical region*:

$$C ::= \ldots \quad | \quad C_1 \;||\; C_2 \quad | \quad \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}$$

The `with` r `when` B `do` C `endwith` implies mutual exclusion of r - that is, no two `with` commands on r may be simultaneously executed, and furthermore waits for B to hold before execution. Conditional critical regions are frequently used to guarantee exclusive access to a mutable shared resource by programmers, and as a result are key to verification of any non-disjoint programs. Here, O'Hearn concentrates on programs with a special form, with resource declarations immediately followed by parallel composition:

$$\begin{array}{l}\texttt{resource}\ r_1\ (\text{variable list}), \ldots,\ r_m\ (\text{variable list}) \\ C_1 \;||\; \ldots \;||\; C_n\end{array}$$

where we declare each protected resource in a declaration which also specifies which variables may be used. A with command protects a resource r which was previously declared, waits for the boolean expression B to evaluate to true, and then once it does, executes C.

Additionally, O'Hearn places some requirements on programs to be 'well-formed':

- A variable belongs to at most one resource, i.e. appears in the variable list of the resource declaration of at most one identifier;
- If a variable belongs to a resource, it may not appear in parallel processes unless inside a critical region for r; and
- If a variable is modified in one process, it must not appear in another unless it belongs to a resource.

These conditions formalize how programmers would consider a shared resource to be protected by a concurrency primitive, as it prevents two threads simultaneously accessing the same variable where at least one is a write unless mutual exclusion is guaranteed by a region.
Each resource has a unique identifier r and resource invariant RI_r, and modifications to it are verified using the following proof rule:

$$\frac{\{(P * RI_r) \wedge B\} C \{Q * RI_r\}}{\{P\}\text{with } r \text{ when } B \text{ do } C \text{ endwith}\{Q\}} \quad \begin{array}{l}\text{No other process modifies} \\ \text{variables free in } P \text{ or } Q\end{array}$$

Shared heap locations can be represented in this model as 'owned' by a resource, instead of a thread, which allows them to be passed between parallel commands in a coarse-grained way. Using the assumption that interleavings of threads may not interleave the bodies of with statements of the same resource identifier, we can use this to verify execution traces of coarse-grained concurrent programs. With this rule, we can use the same parallel rule as given above for disjoint concurrency, with the same side condition that no variable free in P_1 or Q_1 may be modified in C_2 and vice versa - the disjointness of shared regions guaranteed by the semantics of `with` statements suffices to prove the parallel executions are race-free if the previous disjointness conditions are satisfied. As shared heap locations are considered to be owned by the resource context when not in use, this enforces that no part of the heap unprotected by a resource may be shared between threads, while allowing some sharing in the mechanism described above. In combination with the requirements for well-formedness which prevent race conditions with shared variables, we get a sound logic with much more expressivity than previous work.

An important technical detail in CSL proofs is the notion of a 'precise' predicate. In traditional Hoare logic, we have a proof rule for conjunction:

$$\frac{\{P\}C\{Q\} \qquad \{P'\}C\{Q'\}}{\{P \wedge P'\}C\{Q \wedge Q'\}}$$

Without extra conditions on the predicates this rule is unsound in CSL. A predicate is precise if for all states, there is at most one subheap which satisfies the predicate, and therefore there is no ambiguity on which part of the heap the predicate refers to. By considering an imprecise predicate, we can derive different conclusions in our proof trees when considering different parts of the heap, and by allowing the conjunction rule, we can obtain a contradiction. For full details of Reynold's counterexample, see [8], but simply put, O'Hearn's original paper enforces that all predicates be precise in order to ensure soundness, while more modern separation logics often remove the conjunction rule altogether and sidestep the issue.

This model enables proofs of more styles of concurrent programming than previously possible. Crucially, it can be used for mutual exclusion groups represented by semaphores typically in a matching lock and unlock pattern, as well as unmatching patterns where semaphores are used to send signals between processes. Both examples are lifted from [8] where the relevant proofs can be found.

$$P(s) \triangleq \texttt{with } s \texttt{ when } s > 0 \texttt{ do } s := s - 1 \texttt{ endwith}$$

$$V(s) \triangleq \texttt{with } s \texttt{ when true do } s := s + 1 \texttt{ endwith}$$

Standard semaphore encoding with CCR's [8]

```
P(mutex)
critical piece of code
V(mutex)
```

Figure 2.1: An example of 'matching' concurrency from [8]

```
semaphore free:= 1; busy:= 0
...                          ...
P(free);                     P(busy);
10 := m;          ||         n := [10];
V(busy);                     V(free);
...                          ...
```

Figure 2.2: An example of 'unmatching' concurrency from [8]

The matching pattern involves protecting a resource by only modifying shared memory from inside a with command. This is the most basic coarse-grained concurrency pattern where the same lock is used to protect access to an entire resource, and the resource may not be accessed outside of these protected states (enforced by well-formedness of programs). The more interesting unmatching example uses semaphores for two commands to signal each other - the right hand thread has to wait for the left hand thread to raise the *busy* semaphore to 0 before it can proceed, which prevents the left hand thread from reducing the *free* semaphore and subsequently reading the heap location at 10 until the right hand thread has written it. This coordination enforces an ordering on the accesses to the heap location which does prevent race conditions, despite the P and V operations being 'unmatched', and is still verifiable in this CSL.

This logic vastly increased the types of programs verifiable by first introducing concurrent programs, and then accounting for limited forms of interference - precisely those which programmers would usually refer to as *coarse-grained* concurrency. Coarse- and fine-grained concurrency don not have strict, formal definitions, but refer to the notion of 'how much' of a resource is protected by the same primitive. For example, if one lock protects an entire abstract data structure, then a large number of operations occur within one protected region, and there will be a lot of contention for this lock, which is held for longer times. In contrast, fine-grained patterns often involve several locks protecting different components of an abstract data structure with a complex algorithm to lock everything in such a way that each lock indeed protects a consistent component of the structure and their combined use guarantees race-freedom and deadlock-freedom. This results in less contention for any individual lock, and each lock is locked for a shorter period of time, which can often significantly increase performance.

Adjacent to this work, a line of research in more nuanced forms of interference surfaced in the forms of *rely-guarantee* first introduced by Jones in [11] and *deny-guarantee* [12], the first of which was combined with CSL by Vafeiadis as the RGSep logic in [13].

2.3 Rely-Guarantee Reasoning

A major limitation of the Owicki-Gries method [9] is that each application of the parallel rule requires an additional manual proof that each command cannot interfere with the proof of the other, which requires checking that every intermediate assertion is preserved by all atomic actions of the other thread. This becomes unwieldy and makes the rule non-composable, as well as the method non-scalable. Rely-guarantee reasoning is an attempt to rectify this by specifying as a relation precisely what the interference from the environment may be on the thread, and what interference the local thread may have on the environment [11].

In rely-guarantee logics, specifications have the form (P, R, G, Q), where P is a standard Hoare precondition, Q is a two-state predicate postcondition, relating the initial state to the final state, R is a two-state predicate describing the initial and final state after atomic actions by the environment and G is a two-state predicate similarly modelling atomic actions of the program (this is interference for the environment). An action $P \rightsquigarrow Q$ is the effect of some command which transitions the state from one satisfying P to one satisfying Q, and is permitted by R (resp. G) if $(P, Q) \in R$ (resp. $\in G$). Interference is any effect on the state made by another thread. Additionally, in order to be well-formed, it must be the case that the pre- and postconditions P and Q are *stable* under R, that is they are resistant to interference from the environment: $(R; Q) \implies Q$ and $(Q; R) \implies Q$ [13].

2.3.1 RGSep

In his PhD thesis [13], Vafeiadis conjoins traditional CSL with rely-guarantee reasoning to enable verification of concurrent programs with well-defined and controlled interference. Crucial to this is the separation of the logical state into *local* and *shared* states, with a unified assertion language enabling simultaneous reasoning about the two. In RGSep, the syntax of assertions has the following form

$$p, q, r ::= \quad P \mid \boxed{P} \mid p \star q \mid p \wedge q \mid p \vee q \mid \forall x.\ p \mid \exists x.\ p$$

where P, Q are standard separation logic assertions. The first construct, P describes local state, for which interference is forbidden, and must be a traditional separation logic assertion. $\boxed{P}$ is a shared state assertion (and may not be nested). The shared state is *indivisible* - boxed assertions must correspond to entire shared states (although there may be several distinct shared states) - and *shareable*:

$$\boxed{P} \star \boxed{Q} \iff \boxed{P \wedge Q}$$

Instead of the relational specifications introduced in rely-guarantee reasoning, RGSep models interference through actions $P \rightsquigarrow Q$, which describe changes performed to shared states. The semantics of this action is to replace a part of the shared state satisfying P with something satisfying Q. Then the rely and guarantee relations are the reflexive, transitive closure of some set of actions and some action is allowed by R (resp. G) if its effect $P \rightsquigarrow Q$ is contained in R (resp. G). Pre- and postconditions are required in RGSep to be stable, that is, they cannot be invalidated by actions of the other thread. This is the notion that for all states s satisfying separation logic assertion S, and for all actions $(s, s') \in R$, it must be that s' satisfies S also.

Specifications in RGSep are quadruples (p, R, G, q), where p and q are RGSep assertions describing the pre- and postcondition, respectively, and R and G are the rely and guarantee relations setting out precisely the interference caused and tolerated by a thread. The links between the pre- and postcondition are expressed with existentially quantified variables, as common to other separation logics, rather than the relational postcondition seen in traditional rely-guarantee reasoning.

We understand the notion of a command C satisfying a specification (p, R, G, q) to mean: if the program is in a state satisfying p and can rely on the environment not taking actions outside of those in R, then after execution of C the state will satisfy q and will have only taken actions affecting the environment satisfying G. Then $\vdash C$ sat (p, R, G, q).

The key proof rules for RGSep [13]:

$$\frac{\vdash C_1 \text{ sat } (p_1, R \cup G_2, G_1, q_1) \qquad \vdash C_2 \text{ sat } (p_2, R \cup G_1, G_2, q_2)}{\vdash (C_1 || C_2) \text{ sat } (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)} \quad \text{(Par)}$$

The parallel rule clearly enforces that the execution of each parallel command can tolerate the interference from both the environment and the other command, and that the interference it may have is tolerated by the environment.

There are two rules for atomic commmands, the first of which ensures the specification is stable under R and reframes the proof as one of an empty R, and the second of which ensures that the change made $(P \rightsquigarrow Q)$ is allowed under G and asks for a proof of the equivalent non-atomic command. This implicitly frames off the part of the shared state which is not modified, F.

$$\frac{\vdash \langle C \rangle \text{ sat } (p, \emptyset, G, q) \qquad p \text{ is stable under } R \qquad q \text{ is stable under } R}{\vdash \langle C \rangle \text{ sat } (p, R, G, q)} \quad \text{(AtomR)}$$

$$\frac{P,Q \text{ are precise} \qquad \vdash C \text{ sat } (P * P', \emptyset, \emptyset, Q * Q') \qquad (P \rightsquigarrow Q) \subseteq G}{\vdash C \text{ sat } (\boxed{P * F} * P', \emptyset, G, \boxed{Q * F} * Q')} \quad \text{(Atom)}$$

For soundness, the (Atom) rule also requires that all branches of the proof use the same P and Q for the atomic region. This requirement, similarly to the precision requirement seen in traditional CSL ensures that two correct proofs ca not be combined in such a way which introduces a contradiction.
Finally, for primitive commands which do not access the shared state, we have:

$$\frac{\vdash_{SL} \{P\} c \{Q\}}{\vdash c \text{ sat } (P, R, G, Q)} \quad \text{(Prim)}$$

It should be clear that RGSep subsumes both SL and RG-reasoning, as the first is precisely those proofs where the shared state is empty, and the second is precisely the case where the local state is empty. RGSep also subsumes CSL by threading the resource invariant through the assertions, as demonstrated in [13]. It follows that RGSep is strictly more expressive than all the above, and in fact introduces the ability to formally verify *fine-grained* concurrent data structures. An example is given in [13] using a linked list structure with a lock per node, but is much too detailed to reproduce here.

In [13], extensions are given to the above proof system to include procedures (to enable composability), multiple shared regions, and ghost code to enable linearisability based proofs (more on this later). However, there is still a long way to go in achieving true modularity in the notion programmers expect of being able to swap out equivalent implementations, and linearisability proofs are unwieldy with the introduction of ghost code.
From here, I go on to discuss two more key ideas from previous work in software verification which will be combined with the ideas from RGSep to form the basis of the TaDA logic.

2.4 Atomicity and Linearisability

So far, we have considered methods of verification which rely on disjointness of space and the constraint of interference to reason about programs. But it is common for programmers to reason informally in such a way which relies on the disjointness of time to guarantee correctness of their programs. More precisely, we consider certain actions of the program 'atomic', the idea that an effect takes place at a discrete point in time and as such the environment may not observe any intermediate stage between the precondition and the postcondition. Atomicity allows us to construct a total order on the operations of different threads by considering certain actions uninterruptable. In separation logic, we already use this assumption about primitive commands such as reads and writes to the heap, but we do not provide any framework to allow the proof to declare other more complex actions atomic. Commonly, lock-free implementations of data structures are provided by libraries for their improved performance, such as a non-blocking concurrent linked list whose atomic remove might be done by modifying the pointer of the previous element, and then cleaned up afterwards by disposing of the heap location. In a doubly-linked list, a non-blocking concurrent implementation could involve an additional 'removed' flag, whose updating would form the atomic remove while the pointers were modified and heap locations cleaned up later. These structures are guaranteed to be race-free in a concurrent setting due to the atomicity properties they display: the action appears to be instantaneous to clients.

To a certain extent, the shared regions of RGSep allow us to reason about disjointness of time - a lock, which a programmer might use to reason about a thread being the only one at that time with access to some portion of the state, forms a 'shared region', with ownership of the state protected by the lock being transferred between the lock to the lock's owner. As only one thread at a time can own the lock, owning the shared region implies ownership of the lock which implies disjointness of time. However, this is not strong enough to reason about certain synchronisation patterns which do not rely on this notion of ownership transfer to guarantee atomicity.

Alternative verification frameworks have been developed to reason about the disjointness of time - we have a variety of properties in the literature, such as serialisability, sequential consistency and linearisability [14]. We consider these properties in the context of a *history*: a sequence of invocation and response events [15], each with an object, operation, arguments and process name. A subhistory is a subsequence of these with the same ordering. A history is sequential if each invocation is followed by a matching response and each response is preceded by a matching invocation. See that a well-formed history restricted to any process must be sequential, but this is not necessarily the case when restricted to an object.

Lamport defines sequential consistency as "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [16]. A history is sequentially consistent if, for all processes, the ordering of execution of the restriction of the history to the process is contained in the ordering of the original history. This is non-composable, as several concurrent sequentially consistent programs may execute in some interleaving which is not itself sequentially consistent.

Papadimitriou defines serialisability of an execution in [17] as having a result which is equivalent to that of a sequential history of its individual transactions, i.e. each process completes a task before the next can begin, without interference, but not necessarily maintaining the ordering of the original history. serialisability again is non-composable, and some transactions may be required to block.

In contrast, an execution is linearisable as introduced by Herlihy and Wing in [15] if it can be modified to a history by appending zero or more responses and removing invocations with no matching response (and does not violate the semantics of the object) such that it is equivalent to some legal sequential history whose induced ordering contains the induced ordering of the initial execution. The induced ordering on a history is a partial ordering such that two operations are ordered if and only if the first's response comes before the second's invocation. See that this differs from serialisability in two key ways: serialisability does not enforce the ordering of the execution on the equivalent history, and does not allow several operations within the same transaction to be interrupted.

Linearisability has several advantages over correctness conditions from ownership-transfer models and serialisability conditions which make it the most significant existing correctness condition for concurrent atomicity [15]:

- Unlike ownership-transfer models, it allows us to reason about atomicity without relying on invariants of the heap to transfer ownership and guarantee lack of interference.
- Unlike serialisability and sequential consistency, linearisability is composable (or *local*, as referred to by Herlihy): if all the components of a system are linearisable, then the entire system is. However, it is not modular: adding additional components requires re-verifying that all of the existing components are also linearisable with respect to the new one.
- Linearisability can be used to reason about nonblocking concurrency. This is powerful because it opens the door to verifying algorithms for real-time use with optimisations for speed and responsiveness.

This line of research has thus far been completely distinct from separation logic-based frameworks for software verification, and until the introduction of TaDA [3] there was no way to reason about programs which combined resource disjointness and atomicity within the same logic.

2.5 Abstract Predicates

We have discussed how composability and modularity are used to distinguish between the expressiveness and usability of verification frameworks. In this section, we consider work to improve the modularity of proof systems.

Composability is a property of a framework which allows you to prove a property of an object by proving a property of all of its constituent pieces. Alternatively, proving the property for each part of the system does not need to consider outside interference and so has also been referred to in the literature as *locality*. The separation logic-based frameworks that we have seen above, as well as linearisability proofs, are all composable. Composability improves scalability, as very large systems can be verified in smaller pieces and then 'composed'.

In contrast, modularity involves using an appropriate abstraction to define an interface for a module, and verifying that a given implementation satisfies its interface. This promotes code reuse, as well as scalability, as different implementations satisfying the same abstract specification can be swapped within the client, without sacrificing the soundness of verification proofs. This mirrors our expectations when we write code: that any implementation satisfying the interface we specify will result in correct

code for a correct client. The separation logic frameworks we have so far seen are not particularly modular, as specifying the action of a module usually requires a predicate which depends on the underlying implementation. Existing work on context logics [18] allows some fine-grained reasoning on structured data using additional structural connectives and formulae for reasoning about subdata but does not extend to reason about concurrency as naturally as separation logics local reasoning.

Logically different properties of a shared structure using a given abstraction may not be separable in the underlying heap representation, making certain abstractions incompatible with separation logic This is most clearly exemplified by a set specification implemented with a linked list [1], where the links between nodes prevent us from considering nodes separable pieces of data. This means we cannot easily use separation logic to combine proofs about different nodes in the linked list as their underlying representations are not independent. In order to reason about structures like these, concurrent abstract predicates [1] provide the correct level of abstraction of a module specification without depending on the implementation, providing a more modular verification framework. These predicates can specify independent properties even when data is shared between them, and allow us to remove dependencies between the proof of the client and implementation of a module, obtaining a modularity in verification mirroring that which we expect when we swap out implementations of an interface in our code.

2.5.1 Concurrent Abstract Predicates

CAP's [1] key modification to previous CSL's was to adjust the treatment of predicates of shared regions in such a way that provides the 'fiction' of disjointness and therefore permits the usual parallel rule in a sound way without complicated additional side conditions. CAP extends the traditional separation logic's grammar of assertions as follows:

$$P, Q ::= ... \quad | \quad [\gamma(E_1, ..., E_n)]^r_\pi \quad | \quad \boxed{P}^r_I \quad | \quad \alpha(E_1, ..., E_n)$$

A boxed assertion is a CAP assertion about shared region r, with allowed actions on the state $I(\vec{x})$ all being made by primitive atomic commands. CAP enforces that shared regions may not be split, so that all boxed assertions describing region r always describe the whole shared region, which gives rise to the same equivalence as in RGSep: $\boxed{P \wedge Q}^r_I \iff \boxed{P}^r_I * \boxed{Q}^r_I$. The permission assertion $[\gamma(E_1, ..., E_n)]^r_\pi$ declares that the thread has permission π to perform action $\gamma(E_1, ..., E_n) \in I(\vec{x})$ on region r, where π is a fractional permission. This model permits the parallel rule as although shared region assertions can always be duplicated, the actions taken on them are restricted to those specified in the environment and to threads holding the relevant permissions.

The fiction of disjointness given by these regions allows us to specify modules with *concurrent abstract predicates*: each module makes available a number of predicates and axioms which it guarantees for the clients to use in their proofs. Then only in the proof that the module satisfies the specification is the underlying implementation necessary. The final addition to the CAP assertions $\alpha(E_1, ..., E_n)$ are these concurrent abstract predicates, which can be logically manipulated according to the axioms guaranteed by the module. Specifications for modules must be *self-stable*, that is stable under all actions permitted by the module, which in turn means that clients never need to reason about internal interference to a module. This layer of abstraction intuitively corresponds to the way programmers reason about their own code.

$$\frac{\Delta; \Gamma \vdash \{P_1\} C_1 \{Q_1\} \qquad \Delta; \Gamma \vdash \{P_2\} C_2 \{Q_2\}}{\Delta; \Gamma \vdash \{P_1 * P_2\} C_1 || C_2 \{Q_1 * Q_2\}} \text{ Par}$$

Consider the example of a traditional spin lock, lifted from the CAP paper [1]. To begin with, any lock, no matter the underlying representation and implementation, must have functions to a) create a lock, b) lock and c) unlock. The information a client of this module must be able to reason with is if it is able to obtain the lock, and whether it has successfully locked (and therefore owns) it. For this, the lock module defines the abstract predicates $isLock(x)$, and $Locked(x)$, such that the following specifications

and abstract predicate formulae hold:

$$\{isLock(x)\} \quad \text{lock}(x) \quad \{isLock(x) * Locked(x)\}$$

$$\{Locked(x)\} \quad \text{unlock}(x) \quad \{\text{emp}\}$$

$$\{\text{emp}\} \quad \text{makeLock}(x) \quad \{\exists x.\text{ret} = x \wedge isLock(x) * Locked(x) * \circledast_{i=1}^{n}(x+i) \longmapsto \_\}$$

$$isLock(x) \iff isLock(x) * isLock(x)$$

$$Locked(x) * Locked(x) \iff false$$

Figure 2.3: Abstract Lock Specification

These demonstrate the ability of a client to 'share' the knowledge that x refers to a lock, and that it is not possible to lock x twice. Any implementation of a lock module would then have to satisfy this specification.

In order to verify this implementation meets the specification, we must first have some notion of what the underlying concrete representation of the predicates are - I reiterate that this is dependent on the implementation, but is only used in the verification of the module specification, and *not* by the client, and as such we can replace modules with different implementations which satisfy this specification in the client code. In the case of the spinlock, we define the concrete representation of our predicates as

$$isLock(x) \equiv \exists r.\exists \pi.[\text{LOCK}]^r_\pi * \boxed{x \longmapsto 0 * [\text{UNLOCK}]^r_1 \vee x \longmapsto 1}^r_{I(r,x)}$$

$$Locked(x) \equiv \exists r.[\text{UNLOCK}]^r_1 * \boxed{x \longmapsto 1}^r_{I(r,x)}$$

The above isLock definition says that knowing x is a lock gives you the (non-unique) permission to try to lock it, and that the underlying heap location is either zero (in which case the shared region owns full permission to unlock it), or it is one (and some thread(s) own permission to unlock it). The locked predicate means a thread has full permission to unlock the lock (and thus implies this predicate is non-duplicable, as the total permission for an action may not exceed 1) and that the thread can rely on the shared heap location to have the value 1. Actions permitted on a shared region are declared in $I(r,x)$, and specify what allowed actions must satisfy in order to be allowed by a given label (which is associated with a guard).

$$I(r,x) \triangleq \begin{pmatrix} \text{LOCK} : x \longmapsto 0 * [\text{UNLOCK}]^r_1 \rightsquigarrow x \longmapsto 1 \\ \text{UNLOCK} : x \longmapsto 1 \rightsquigarrow x \longmapsto 0 * [\text{UNLOCK}]^r_1 \end{pmatrix}$$

```
lock(x) {
  local b;
  do ⟨b := ¬CAS(&x, 0, 1)⟩
  while(b)
}

unlock(x) {
  ⟨[x] := 0⟩
}

makelock(n) {
  local x := alloc(n + 1);
  [x] := 1;
  return x;
}
```

Figure 2.4: Implementation of a CAS-based spinlock [1]

We can see intuitively that the above implementation satisfies the specification we have given: makelock allocates memory, set the location representing the heap to 1, i.e. locks it, and then returns. The proof relies on the idea that it is always possible to make a new shared state from a piece of heap that you own. It is also fairly easy to see that unlock satisfies its specification - by holding the UNLOCK permission, we may take this atomic action (atomic actions are surrounded by $\langle..\rangle$), after which the thread no longer owns the lock. Meanwhile, lock 'spins' on the while, attempting to atomically update the lock from 0 to 1. If this fails, it was because the lock is held elsewhere, and we continue spinning until we succeed, in which case we can guarantee the cell had value 0 when we succeeded, and held the UNLOCK permission. After the successful CAS, the thread holds the UNLOCK permission and can guarantee the heap location is 1, so we can return *Locked(x)* as well as the *isLock(x)* we already had.

CAP gives us reasoning about fine-grained concurrency with the abstractions necessary to have a truly modular proof system, but does not incorporate any reasoning about linearisability. We will see TaDA, which uses all the ideas we have discussed so far to construct an expressive logic for many types of concurrent programs.

Chapter 3

TaDA and TaDALive

3.1 TaDA

The original TaDA paper [3] made two primary contributions: atomic triples, to specify logical atomicity, and a program logic able to handle both separation based and linearisability based verification of concurrent programs.

3.1.1 Atomic triples

The introduction of concurrent abstract predicates briefly presented in Section 2.5 allowed us to write our specifications at a higher level of abstraction than previously, separating the specification of a module from its implementation. Analogously to this, abstract atomic triples will allow us to specify an atomic action at a higher level of abstraction than program execution, so that we can reason about atomic actions whose implementation is more complex than a primitive command - the core idea behind 'logical atomicity'. An atomic triple judgement,

$$\vdash \langle p \rangle \ \mathbb{C} \ \langle q \rangle$$

reads as: when starting in a state satisfying separation logic assertion p, the program $\mathbb{C}$ and the environment both preserve p (although the underlying implementation of the state may change) until the atomic update happens, and immediately following the atomic update the state must satisfy q. Once the atomic update has happened, the environment is no longer under any obligation to preserve q and the thread may not assume any ownership of the resources [3]. The atomicity of the program is inherently dependent on the abstraction level of the predicates, as an environment observing the memory from a 'lower' abstraction level may observe intermediate steps and thus the $\mathbb{C}$ would no longer represent an atomic action. This is similar to how the introduction of a new non-atomic function can break a linearisability proof.

The natural next question to ask is how to use these atomic triples to write specifications for more complex code: what if some parts of the concrete state are updated atomically, but other parts are not? And how do we express interference on the atomically updated precondition, when it is clear that the interference allowed by the environment is much less restricted than the thread itself (as the thread itself must act on the atomically updated state represented by the atomic precondition at most once)?
Let us consider the following implementation of a spinlock: we aim to write a specification for $lock(x)$.

```
def lock(x) {
  var d = 0 in
    while (d = 0)
      d := CAS(x, 0, 1)
}
```

```
def unlock(x) {
  [x] := 0
}
```

```
def makelock() {
  var x = 0 in
    x := new(1) ;
    [x] := 0 ;
    ret := x
}
```

Figure 3.1: Spinlock implementation

The specification verified in CAP is

$$\{\text{isLock}(\mathtt{x})\}\ \text{lock}(\mathtt{x})\ \{\text{isLock}(\mathtt{x}) \star \text{Locked}(\mathtt{x})\}$$

(as in Example 2.3). To any programmer, locking a lock is considered an atomic action - it is crucial in order to guarantee mutual exclusion that the locking action happens when a lock is unlocked and without another thread interrupting in between to also lock it. In order to turn this into an atomic specification, we consider some potential specifications (written in terms of their underlying heap representations).

$$\langle \exists l.\ \mathtt{x} \mapsto l \rangle \text{lock}(\mathtt{x}) \langle \mathtt{x} \mapsto 1 \rangle$$

However, if the lock were already locked, this would allow the lock method to simply return without taking action - this clearly does not guarantee mutual exclusion. Attempting to overcome this could lead to the following specification:

$$\langle \mathtt{x} \mapsto 0 \rangle \text{lock}(\mathtt{x}) \langle \mathtt{x} \mapsto 1 \rangle$$

This is more subtly wrong - in a number of ways. For example, this assumes the environment will not change the state of the lock (so it is not shareable between threads and therefore defeats the purpose of the lock), and furthermore still does not specify how the lock should behave if we want to acquire the lock while it is held by another thread.

All of these issues, more ideologically, are caused by a fundamental imbalance in the way environment interference can be tolerated on atomically updated objects: we expect other threads to also be acquiring and releasing the lock, and our specification needs to be explicitly dependent on the state of the lock at the time of the action (linearisation point) due to this environment interference, not just on the state before the method was called. TaDA introduces a semantically new quantifier, $\mathbb{V}$, to bound logical variables in such a way to capture this interference.

$$\mathbb{V} l \in \{0, 1\}.\ \langle \mathtt{x} \mapsto l \rangle \text{lock}(\mathtt{x}) \langle \mathtt{x} \mapsto 1 \star l = 0 \rangle$$

This pseudoquantified logical variable l describes the following semantic meaning: the environment is free to change l between 0 and 1 as many times as it wishes, as long as it does not change l outside the bounds of the quantification, but the local thread may not change the value of l. At the linearisation point of the code, the value of l is fixed (because this is an atomic action, so the environment may not interrupt it to modify the value of l), and furthermore at the linearisation point, according to the established postcondition, the lock was unlocked ($l = 0$), so regardless of the state of the lock initially, the local thread did indeed lock the lock itself.

This explicit control of logically atomic actions in the proof system allows us to verify the implementation in Figure 3.1, which differs from implementation Figure 2.4 in the previous chapter, notably in the missing angled brackets around the atomic actions. Before TaDA, it was frequently necessary to include this 'ghost' construct in the commands where the atomicity of commands such as CAS was essential to the race-freedom of the verified code. This would tell the proof system "this code inside the angled brackets is implemented atomically and cannot be interrupted! Please just believe me!", in order to dispatch additional technical requirements of the proof. The absence of this ghost code is the simplest way to observe the key contribution of being able to do proofs relying on logical atomicity within the logic itself.

We proceed from here, continuing with the example of the spinlock, to provide intuition on the key introductions in TaDA's program logic in order to first prove code is logically atomic, and second to use these proofs to verify that client code does not contain race conditions.

3.1.2 Program Logic

Keeping in mind that TaDA is designed to be a self-contained program logic with logical atomicity proofs an *additional* feature to existing separation logic based proofs, we need to combine Hoare specifications and the RGSep-style shared regions and interference to maintain the ability to verify code which really is fundamentally about ownership, as well as the genuinely new technical machinery to reason about atomicity.

Specification language. TaDA's specifications use assertions on both locally owned resources with well-defined interference protocols and permissions (guards), and atomically updated resources which are

usually shared and require additional restrictions on the interference permitted from the environment. Consider briefly the program

$$\mathtt{x} := \mathtt{FAS}(\mathtt{y}, 1)\ ;\ \mathtt{z} := \mathtt{x} + 1 \tag{3.1}$$

This contains an atomic action - a FAS instruction - and could be embedded within a larger piece of code which requires the update on the part of the heap at $\mathtt{y}$ to be atomic - perhaps this $\mathtt{y}$ represents a lock, or other low-level flag for thread communication. We might require this action to be atomic in order for our program to be race-free, and yet it is clear the program as a whole is not atomic, as it contains two sequenced commands (with the potential for interruption between). To specify and verify such programs, TaDA combines traditional Hoare triples with atomic triples to produce *hybrid* specifications.

$$\mathbb{V}x \in X.\langle p_p \mid p(x)\rangle \quad \cdot \quad \exists \mid\ y \in Y \langle q_p(x,y) \mid q(x,y)\rangle$$

As discussed, this contains pseudoquantified logical variable x, a locally owned, *private* precondition p_p and an atomically modified (potentially with shared ownership) precondition $p(x)$, dependent on the pseudoquantified x, allowing us to express the different requirements of local interference versus environment interference on the shared part of the resources. As in some cases the result of the postcondition could be nondeterministically dependent on the state of the resources at the linearisation point, we have an additional pseudoexistentially quantified variable y which ties together the two parts of the postcondition. Similar to the precondition, we have a locally owned, *private* postcondition $q_p(x,y)$, which may still depend on the values produced at the linearisation point, and the atomic postcondition $q(x,y)$. As discussed in the previous section, the atomic postcondition is established at the linearisation point, after which the environment interference is no longer controlled and therefore may be immediately externally modified. Therefore the local thread may not assume it continues to hold any ownership of the resources in the atomic postcondition (unless some ownership transfer has occurred, as is frequent in separation logic based proofs), and these resources may henceforth be destroyed or modified in unspecified ways. This hybrid specification is expressive enough to specify Example 3.1, allowing for environment interference on the heap location at $\mathtt{y}$ and capturing the requirement that $\mathtt{z}$ is not updated as part of the atomic action:

$$\mathbb{V}n \in \mathbb{Z}.\langle\ \mathsf{emp} \mid \mathtt{y} \mapsto n\ \rangle \quad \mathtt{x} := \mathtt{FAS}(\mathtt{y}, 1)\ ;\ \mathtt{z} := \mathtt{x} + 1\ \langle \mathtt{z} = \mathtt{x} + 1 \mid \mathtt{y} \mapsto 1 \star \mathtt{x} = n\rangle$$

Shared regions and interference. TaDA's handling of permission-based interference on shared resources is in spirit very similar to RGSep and CAP, although the technical machinery looks a little different. Explicitly, instead of boxed assertions with their own region label r and interference protocol I, TaDA requires a globally defined set of shared region types and unique region identifiers, so we may have many instances of shared regions corresponding, for example, to a spinlock region type **spin**, each with a unique region identifier r. Then there are semantic and syntactic region interpretations: each region type has a well-defined semantic interpretation in the model, as well as an underlying syntactic assertion representing it. Concretely, for a spin lock, our syntactic interpretation of $\mathbf{spin}_r(\mathtt{x}, l)$ might be $\mathtt{x} \mapsto l$, while the semantic interpretation must correspond exactly with the semantics of this assertion. This allows us to use the assertion language of shared regions in order to control interference, while having explicit control of the concrete underlying representation in order to verify code modifying it. As in CAP, shared regions are shareable and should not be seen as their concrete underlying representation without further machinery to guarantee exclusive control: $\mathbf{t}_r \iff \mathbf{t}_r \star \mathbf{t}_r$ holds.

As is common among concurrent separation logics, TaDA is parameterised by a set of guards, which are a construct of the assertions, and for each shared region, the guards associated with that region form a partial commutative monoid with composition induced by the separating conjunction. Then, a labelled transition system for each region type $\mathbf{t}$, $\mathsf{Guard}_{\mathbf{t}} : \mathsf{AState} \rightsquigarrow \mathsf{AState}$ defines precisely which actions are permitted by a guard. These guards fill the role of both rely *and* guarantee: we can rely in our proofs on the environment not making modifications to shared regions for which it does not own an appropriate guard, thus if a thread owns an exclusive guard, or there does not exist a compatible guard with that owned by the thread, then the environment cannot make certain updates to shared regions. This encapsulates much of the rely-guarantee reasoning for shared regions when the proof does not depend on the notion of logical atomicity. This makes TaDA at least as expressive as these existing proof systems.

The final technical detail required is region levels: as our shared regions are encapsulated in region assertions $\mathbf{t}_r^{\lambda}(x)$ and shareable, we need to prevent inconsistencies in the logic caused by opening the same shared region twice - for example, $\mathbf{spin}_r(\mathtt{x}, l) \star \mathbf{spin}_r(\mathtt{x}, l)$ is a sensible assertion equivalent to $\mathbf{spin}_r(\mathtt{x}, l)$, while the region interpretation $\mathtt{x} \mapsto l \star \mathtt{x} \mapsto l$ is obviously not well-defined. The issue is side-stepped by

parameterising shared regions with a level from some well-founded order (typically $\lambda \in \mathbb{N}$), and in the proof rules including a region level in the context controlling which regions may be opened and modified.

Atomicity contexts and tracking resources. TaDA introduces a new type of ghost state to fuse logical atomicity based proofs into the underlying logic. There are two primary goals: verify that code which is not a primitive atomic command is logically atomic, and combine these proofs with the underlying proof system in such a way which allows us to verify atomic actions without the usual exclusivity requirement guaranteeing race-freedom.

Before discussing examples, I provide an overview of the new constructs. There are two new components: the atomicity context $\mathcal{A}$, and the atomicity tracking resource $r \Mapsto d$, where d represents an element of $\{\blacklozenge, \lozenge\} \uplus (\mathsf{AState} \times \mathsf{AState})$. The atomicity context keeps information in the proof context about which regions we are verifying some action is logically atomic, the abstract state the region may be in (initially taken from the pseudoquantified logical variable), and the abstract state the action must establish. It prevents us from trying to nest atomic actions (which would obviously lose their atomicity), as well as determining which concrete changes do and do not represent abstract state changes, with respect to the abstraction level of the initial predicate.
The atomicity tracking resource is an assertion which takes the place of a guard during proofs of logical atomicity, i.e. provides 'permission' to update the shared resource, but also serves as a proof certificate that the update occurred atomically and of the initial and resulting state at the linearisation point was. The $\blacklozenge$ refers to an atomic action which the thread has the responsibility to take, the $\lozenge$ bears witness to the linearisation point, and elements of $\mathsf{AState} \times \mathsf{AState}$ refer to the abstract state immediately before the linearisation step and the resulting state.

Example of logical atomicity proof. Return to the locking of a spinlock, using the implementation in 3.1, reproduced below.

```
def lock(x) {
  var d = 0 in
    while (d = 0)
      d := CAS(x, 0, 1)
}
```

I use the machinery in the original TaDA paper for the following example, although this differs from the machinery used to prove an equivalent specification in the TaDALive paper and the eventual proof in this thesis.
For a shared region $\mathbf{spin}_r(\mathtt{x}, l)$ with interpretation $\mathtt{x} \mapsto l$, and a single non-zero guard G with transition system

$$G : 0 \rightsquigarrow 1$$
$$G : 1 \rightsquigarrow 0$$

We aim to check

$$\forall l \in \{0, 1\}\ \langle\ \mathbf{spin}_r(\mathtt{x}, l) \star [G]_r\ \rangle\ \mathrm{lock}(\mathtt{x})\ \langle\ \mathbf{spin}_r(\mathtt{x}, 1) \star l = 0 \star [G]_r\ \rangle$$

The atomicity proof rule which allows us to prove $\mathrm{lock}(\mathtt{x})$ happens atomically is MAKE ATOMIC:

$$\frac{r \notin \mathcal{A} \qquad \{\ (x, y) \mid x \in X, y \in Q(x)\ \} \subseteq \mathcal{T}_{\mathbf{t}}(G)^* \qquad \lambda', r : x \in X \rightsquigarrow Q(x), \mathcal{A} \vdash \{p_p \star \exists x \in X.\ \mathbf{t}_r^{\lambda}(x) \star r \Mapsto \blacklozenge\}\ \mathbb{C}\ \{\exists x \in X, y \in Q(x).\ q_p(x, y) \star r \Mapsto (x, y)\}}{\lambda', \mathcal{A} \vdash \forall x \in X.\ \langle\ p_p \mid \mathbf{t}_r^{\lambda}(x) \star [G]_r\ \rangle\ \mathbb{C}\ \exists y \in Q(x).\ \langle\ q_p(x, y) \mid \mathbf{t}_r^{\lambda}(y) \star [G]_r\ \rangle}$$

At a general level, this proof rule should be read: $\mathbb{C}$ is safe and abstractly atomic if the update it makes to the shared region $\mathbf{t}_r^{\lambda}(x)$ is permitted by the owned guard G, we are not already in the midst of a proof of some other atomic action on the same shared region r, and we can prove that this command is safe and performs the action in the specification (in a non-atomic way), with the atomicity tracking resource present to bear witness that the abstract state was only locally modified once from x to y. Observe that the guard is replaced in the premise with the atomicity tracking assertion, preventing the guard from justifying additional changes to the shared region.

In our example, the guard $[G]_r$ in the precondition labels the transition $0 \rightsquigarrow 1$, so is sufficient to give permission to lock the shared region.

Moving forward, approximately, we aim to prove the following, where

$$L = \{0, 1\}$$
$$Y = \{1\}$$
$$\mathbb{C} = \texttt{while}\ (\texttt{d} = 0)\ \texttt{d} := \texttt{CAS}(\texttt{x}, 0, 1)$$

$$\lambda', r : l \in L \rightsquigarrow Y \vdash \{\texttt{d} = 0 \star \exists l \in L.\ \mathbf{spin}_r(\texttt{x}, l) \star r \Rightarrow \blacklozenge\ \}\ \mathbb{C}\ \{\ \exists l \in L, y \in Y.\ l = 0 \star y = 1 \star r \Rightarrow (l, y)\ \}$$

Following the application of MAKE ATOMIC, usual proof rules such as WHILE and CONS can be applied to the Hoare specification in the premise of MAKE ATOMIC. This is not substantially different to a traditional derivation. The remaining key to the puzzle is how to dispatch the atomic update: how to perform the update using the atomicity tracking resource as permission, and update the certificate to reflect the values at the linearisation point. We need one more atomicity rule for this, UPDATE REGION:

$$\frac{\lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\ \langle\ p_p \mid I(\mathbf{t}_r^\lambda(x)) \star p(x)\ \rangle\ \mathbb{C}\ \exists y \in Y.\ \left\langle\ q_p(x,y)\ \middle|\ \begin{array}{l}\exists z \in Q(x).\ I(\mathbf{t}_r^\lambda(z)) \star q_1(x,y) \vee \\ I(\mathbf{t}_r^\lambda(x)) \star q_2(x,y)\end{array}\ \right\rangle}{\begin{array}{l}\lambda + 1, r : x \in X \rightsquigarrow y \in Y(x), \mathcal{A} \vdash \mathbb{V}x \in X, \quad \langle\ p_p \mid \mathbf{t}_r^\lambda(x) \star p(x) \star r \Rightarrow \blacklozenge\ \rangle \\ \qquad \mathbb{C} \\ \qquad \exists y \in Y.\ \left\langle\ q_p(x,y)\ \middle|\ \begin{array}{l}\exists z \in Q(x).\ \mathbf{t}_r^\lambda(z) \star q_1(x,y) \star r \Rightarrow (x,z) \vee \\ \mathbf{t}_r^\lambda(x) \star q_2(x,y) \star a \Rightarrow \blacklozenge\end{array}\ \right\rangle\end{array}}$$

Update region allows us to open up a shared region assertion into its interpretation to be manipulated by command. Observe that this results in a reduction in the region level from $\lambda + 1$ to λ, where λ is the level of the region. This prevents us from duplicating the shared region assertion and then opening it twice. The atomicity tracking context allows us to both modify some concrete part of the shared region, while maintaining the abstract state (and thus maintaining the $\blacklozenge$), or performing an atomic update (such as the CAS operation) and updating the abstract shared state, witnessed by an update of the atomicity tracking resource to (x, z).

The usage of both proof rules is for locking a spinlock is demonstrated in Figure 3.2 as per the TaDA paper [19].

$$\begin{array}{l}\mathbb{V}l \in \{0,1\} \vdash \\ \quad \langle \mathbf{spin}_r^\lambda(\texttt{x}, l) \star \lceil G \rceil_r \rangle \\ \quad \left|\begin{array}{l}\text{make atomic} \\ \left|\begin{array}{l}\text{update region} \\ \left|\begin{array}{l} r : y \in \{0,1\} \rightsquigarrow 1 \wedge y = 0 \vdash \{\exists y \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\texttt{x}, l) \star r \Rightarrow \blacklozenge\} \\ \texttt{while}\ (\texttt{b} = 0) \\ \{\exists y \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\texttt{x}, l) \star r \Rightarrow \blacklozenge\} \\ \quad \left|\begin{array}{l}\langle \texttt{x} \mapsto y \rangle \\ \texttt{b} := \texttt{CAS}(\texttt{x}, 0, 1) \\ \left\langle \begin{array}{l}(\texttt{x} \mapsto 1 \wedge y = 0 \wedge b = 1) \vee (\texttt{x} \mapsto \\ l \wedge l \neq 0 \wedge \texttt{b} = 0)\end{array} \right\rangle \end{array}\right. \\ \left\{\begin{array}{l}\exists y \in \{0,1\}.\ \mathbf{spin}_r \texttt{x}, y \star \\ (r \Rightarrow (0,1) \wedge b = 1) \vee (r \Rightarrow \blacklozenge \wedge b = 0)\end{array}\right\} \end{array}\right. \end{array}\right. \\ \quad \langle \mathbf{spin}_r^\lambda(\texttt{x}, 1) \star \lceil G \rceil_r \star l = 0 \rangle \end{array}$$

Figure 3.2: Proof sketch for spin lock, as per TaDA [19]

Applying logical atomicity proof. The purpose of verifying that actions happen atomically is to verify client code which relies on linearisability arguments is safe and correct. In order to apply these arguments, TaDA provides a USE ATOMIC rule.

$$\frac{r \notin \mathcal{A} \qquad \forall x \in X.\ (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(G(x))^* \qquad \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\ \langle\ p_p \mid I(\mathbf{t}_r^\lambda(x)) \star p(x) \star [G(x)]r\ \rangle\ \mathbb{C}\ \exists y \in Y.\ \left\langle\ q_p(x,y)\ \middle|\ I(\mathbf{t}_r^\lambda(f(x))) \star q(x,y)\right\rangle}{\lambda + 1, \mathcal{A} \vdash \mathbb{V}x \in X,\ \langle\ p_p \mid \mathbf{t}_r^\lambda(x) \star p(x) \star [G(x)]r\ \rangle\ \mathbb{C}\ \exists y \in Y.\ \left\langle\ q_p(x,y)\ \middle|\ \mathbf{t}_r^\lambda(f(x)) \star q(x,y)\ \right\rangle}$$

See how this rule allows us to make changes to shared resources without enforcing the exclusive ownership ideas from separation logic. Furthermore, the atomic weakening rule

$$\frac{\lambda, \mathcal{A} \vdash \forall x \in X,\ \langle\ p_p \mid p(x) \star p'\ \rangle\ \mathbb{C}\ \exists y \in Y.\ \left\langle\ q_p(x,y) \;\middle|\; q(x,y) \star q'(x,y)\ \right\rangle}{\lambda, \mathcal{A} \vdash \forall x \in X,\ \langle\ p_p \star p' \mid p(x)\ \rangle\ \mathbb{C}\ \exists y \in Y.\ \left\langle\ q_p(x,y) \star q'(x,y) \;\middle|\; q(x,y)\ \right\rangle}$$

allows us to use atomicity of updates to shared regions to verify the safety of non-atomic programs.

3.1.3 Evaluation

The original TaDA paper [3] had two significant novel contributions - the introduction of atomic triples to reason about logical atomicity, and a program logic facilitating proofs based on separation logic and/or linearisability within the same proof system. These contributions expand the set of verifiable programs from previous logics to include those requiring a linearisation argument, which is often the case for heavily optimised nonblocking concurrent code. Furthermore, by integrating linearisability proofs smoothly with existing work on ownership proofs, TaDA is able to maintain the expressivity of RGSep and CAP in handling fine-grained locking protocols. TaDA achieves these things while maintaining the modularity properties that CAP [1] introduced, as it facilitates abstract predicate use in the proof rules and semantic model of the program logic, thus ensuring that proofs are independent of the implementation of underlying modules. This promotes proof reuse almost to the scale at which programmers require code reuse.

The published literature on TaDA, including the extended technical report [19] and PhD thesis [20], is, in places, imprecise and incomplete.

- The example proofs frequently do not define necessary things like the operation on guards, and some applications of the proof rules fail to satisfy necessary assumptions. There are also a number of misleading or incorrect statements, notation or assumptions, like the pseudoexistential quantifier in TaDA which has no additional meaning to a typical existential, and is replaced in TaDALive with the usual $\exists$ symbol.
- There is an incorrect and generalised stability condition imposed on the predicates in specifications describing shared resources, which in practice almost no specifications with meaningful atomic behaviour satisfy
- Necessary side conditions for soundness on proof rules are frequently omitted (including non-trivial stability assumptions).
- The existing semantic model is ad-hoc, without a coherent structure to reduce the complexity to a tractable object, and is too imprecise to be able to find sources of unsoundness in the logic. Its complexity makes it hard to extend for further work, and furthermore, the author, in the conference paper, presents an expectation that the semantic model is insufficient to verify more complex concurrent programming patterns, such as helping or synchronization, and further work is required in order for TaDA to be extensible in this direction.

I revisit these points in Chapter 8 to discuss how I have addressed them.

3.2 TaDALive

Prior to the introduction of TaDALive [6], there existed a scattering of work on termination properties of nonblocking concurrent code, often in conjunction with safety properties in the usual rely-guarantee style, such as Total-TaDA [21] and Lili [22]. By nonblocking, we refer to algorithms in which the termination of a thread does not depend on the actions of another - this is referred to in Total-TaDA as *non-impedance*. It maintains the existing composabability of proofs (as the termination reasoning is thread-local) as well as the ability to specify programs at different levels of abstraction, but the reality of fine-grained concurrent programs is that they frequently employ a variety of busy-waiting style techniques for synchronization, and usually it is the case that for example the successful acquiring of a locked lock is dependent on another thread first releasing the same lock. To complicate things further, when this busy-waiting takes execution steps without making any logical progress, there's no sensible logical encoding of the execution steps in such a well-founded way which would guarantee termination: TaDALive calls this pattern of taking steps without making progress *abstract atomic blocking*. TaDALive makes a number of significant contributions to break into these complex fine-grained termination patterns in such a way which is thread-local (thus composable), and expressed at the correct abstraction level (so as not to lessen proof reuse).

3.2.1 Liveness

Similar to the way rely-guarantee reasoning can be seen as *invariant* based - threads and the environment must always act within their bounds - TaDALive's fundamental idea is to base progress proofs on *liveness invariants* - guarantees that a property will always eventually hold - and proposes that its ubiquitousness in the way programmers reason about their own code supports the idea that this could be considered a definition of abstract blocking, or at least a fundamental property of it. As in many modern separation logics which have logical ghost state in the forms of guards to express rely-guarantees, TaDALive introduces a novel form of logical ghost state, termed *subjective obligations*, which encode liveness invariants in assertions. These are used in a thread-local way to encode both obligations which the environment requires the thread to fulfill and obligations which the thread depends on the environment to eventually fulfill in order to guarantee termination, although the composition of environment and local obligations is rather subtly defined. This locality gives rise to a proof system which maintains composability of earlier logics, with the parallel rule requiring very little additional machinery to be sound, except for a technical detail to prevent unsound circular reasoning which is in practice easily dispatched.
As in the environment interference assumption $\forall x \in X$ of TaDA, where the safety of a program is conditional on additional restrictions to the environment interference, TaDALive introduces environment liveness assumptions for specifications similar to TaDA's, written

$$\forall x \in X \twoheadrightarrow X'.\ \langle\ P_h \mid P_a(x)\ \rangle\ \cdot\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle$$

Observe that the structure of TaDA's hybrid specifications is kept except for minor notational changes such as p_p to P and the replacing of the pseudo-existential with a traditional existential which better reflect the semantics. The key introduction is the environment liveness assumption $\forall x \in X \twoheadrightarrow_k X'$, which should be read as the environment interference on the abstract state x is restricted to modifications within X, and furthermore, that the environment will always eventually update x to satisfy X'.

The culmination of all this work is the ability to provide *total* specifications for spinlocks, CLH locks and a variety of fine-grained client code.

3.2.2 Semantic Model

The new semantic model I present in this work is closely based on that of TaDALive's, so I provide an overview of the TaDALive semantic model, to provide the reader with some intuition in understanding how the components of my semantic model fit together later.

Verification of liveness invariants of environment steps requires the semantic model to be able to reason about environment steps at the resolution of one concrete execution step at a time - simply observing the state before and after each logical step is insufficient, as it may be that the environment establishes its obligations in an intermediate step and then subsequently violates them. Intuitively, this should still satisfy the environments obligations, but needs to be observable by the model to be correctly specified in the semantics. As a result, TaDALive's semantic model is a novel trace-based design, with tight control over the logical ghost state permitted and tracked in each step of the trace. This provides precise reasoning about environment steps at the resolution of one concrete execution step at a time and is the only way to guarantee that liveness assumptions are not missed. TaDALive has four different layers of semantics, separating the concerns of different parts of the model, each with complex interactions.

The trace represents the underlying machine represention of a program. A program trace represents the program execution of a program - similar to a trace, but with an additional component for the program being executed. Then, the logical state (worlds) includes a heap as well as each type of ghost state so that we can reason about resources at a more abstract level, and crucially are thread-local, with composition giving us the perspective from several threads, and eventually the entire system. These semantics gives rise to specification traces, whose states are logical instrumentations of the concrete resources (worlds) and transitions checked against the rely and guarantee. Finally, world traces are used to verify liveness properties. These clear boundaries between abstraction levels make it easier to reason about each component, and the ultimate safety check is disjoint from the liveness check, which makes these components more separable. The soundness judgement is defined by checking that all program traces representing an execution of a given command have a safe logical instrumentation which satisfies the specification, i.e. the initial state satisfies the precondition, if it terminates then it satisfies the postcondition. The safety judgement is based on alternating data and determines whether each step is permitted by the logical instrumentation, with environment steps represented by universally branching transitions, and local steps by existentially branching transitions.

3.2.3 Evaluation

TaDALive improves greatly on the variety and complexity of programs verifiable with total specifications, with a program logic that can handle liveness arguments dependent on other threads behaviour without sacrificing on composability. With novel ghost states, TaDALive allows threads to express their dependencies on the environment as subjective obligations, maintaining some modularity properties. The environment liveness assumptions replace TaDA's environment interference assumption while maintaining TaDA's reasoning about abstract logical atomicity, so that they can be expressed at the correct level of abstraction. In achieving these goals, TaDALive's new semantic model is completely different to TaDA's, with highly complex interactions between several types of logical ghost state handled in such a way that each level of abstraction is self-contained and easier to work with, in comparison to the TaDA semantic model which has little separation of elements of different abstraction levels.

In making such significant changes, the complexity of TaDALive is such that it is very difficult for one person to be able to understand and connect together all of the pieces.

- The complexities of these liveness arguments are such that the semantic model is intractably large, which makes it very difficult to extend further to verify other complex concurrency patterns such as helping or synchronization. TaDALive does not currently improve on the safety verifications of TaDA.
- The semantic model does not provide any definition or integration of abstract predicates, reducing modularity compared to TaDA and earlier work.
- In fixing the problems with the original TaDA logic, TaDALive skates over more established technical details, which in a few places, results in more technical complexity than is strictly necessary and definitions which are not well-defined (although in comparison to TaDA, it is clear what the intention is). Given the overwhelming size of the logic, this is not a surprise, but makes it necessary to find genuine simplications to the model before it will be possible to extend to other concurrency patterns.
- Some of the issues with missing technical details are easily resolved, but notably explicit handling of some parts of the logical state is missing, and the proof rules of some primitive commands are not well-defined either.

Again, a full comparison of how TaDA 2.0 addresses these concerns is in Chapter 8.

3.3 Related Work

Lili Lili [22] is a predecessor to TaDALive which was also intended to introduce reasoning about liveness properties to modern concurrent separation logics, and like TaDALive allows for linearisability proofs of safety properties as well as liveness properties within the same logic under the assumption of fair scheduling. Lili uses a completely different meta-theory to TaDALive, with contextual refinements providing abstractions for concurrent objects. However, Lili does not have a parallel rule and thus cannot be used to verify much of the concurrent programs TaDALive does. It also makes use of global ghost state, which is less modular than TaDALive's subjective obligations.

Iris The line of research which has surfaced as the major player in software verification in recent years is Iris, a language-independent framework for reasoning about the safety of concurrent programs, with a Coq implementation and instantiation into several different programming languages. Iris was inspired by TaDA and consequently is built to accommodate ownership and linearisability based proofs. A large amount of work has been put into simplifying the underlying concurrent separation logic into a number of key constructs, in order for the framework to be appropriate for a wide variety of problems - this aims to solve the '700 separation logics' problem described by Matthew Parkinson in [23], that inventing a new separation logic for each individual library and language was becoming commonplace and unfeasible to continue. Iris seems thus far to be extensible in such a way to handle the variety of complex problems asked of modern separation logics, including helping and synchronization, but the underlying logic does contain a number of features not seen in traditional separation logics including a number of *modalities*, and a higher-order assertion language [24]. It is likely that these features are why Iris has become so easily extensible and therefore widely used, but raises the question of whether they are actually necessary to solve certain problems. Specifically, the consolidation and continued work on TaDA and TaDALive aims to handle more complex concurrent interactions without Iris' higher-order state.

Furthermore, in order to resolve soundness concerns with impredicative invariants, Iris uses a step-indexed semantics [25] with a later modality to prevent circular dependencies. Spies et al. later determined that this step-indexed semantics is insufficient to prove liveness properties and use transfiniteness to introduce reasoning about termination to Iris in [26], but is not sufficient for concurrent programs.

Trillium Trillium [27] is a brand-new framework which uses Iris to establish intensional refinements to strengthen concurrent separation logics to accommodate for liveness properties, and is the first to verify concrete implementations of distributed protocols such as Paxos are correct with respect to their abstract TLA+ specifications. It hopes that applying Trillium to Transfinite Iris it would resolve some current limitations, and bring the expressivity required to verify preservation of liveness properties. However, it is ultimately for expansion into liveness proofs of Iris, rather than understanding how to verify safety properties of more complex code with less complex logics.

Chapter 4

TaDA 2.0 - Syntax and Semantic Model

Both the program and specification language range over the following:

$$\begin{aligned}
\mathtt{b} \in \mathsf{Bool} &\triangleq \{\mathtt{true},\ \mathtt{false}\} \\
\mathtt{v} \in \mathsf{Val} &\triangleq \mathbb{Z} \cup \mathtt{Bool} \\
\mathbb{E} \in \mathsf{Exp}(\mathsf{Vars}, \mathsf{Vals}) &::= v \mid x \mid \mathbb{E}_1 + \mathbb{E}_2 \mid \mathbb{E}_1 - \mathbb{E}_2 \mid \ldots \\
\mathbb{B} \in \mathsf{BExp}(\mathsf{Vars}, \mathsf{Vals}) &::= b \mid x \mid \neg\mathbb{B} \mid \mathbb{E}_1 \leq \mathbb{E}_2 \mid \ldots
\end{aligned}$$

4.1 Commands

Our commands range over program expressions (and therefore program booleans and program values), program variables, and $f \in \mathsf{FName}$ a set of function names.

$$\begin{aligned}
\mathtt{x} \in \mathsf{PVar} &\triangleq \{\mathtt{x}, \mathtt{y}, \ldots\} \cup \{\mathbf{ret}\} \\
\mathtt{E} \in \mathsf{PExp} &\triangleq \mathsf{Exp}(\mathsf{PVar}, \mathsf{Val}) \\
\mathtt{B} \in \mathsf{PBExp} &\triangleq \mathsf{BExp}(\mathsf{PVar}, \mathsf{Val})
\end{aligned}$$

Definition 4.1.1 (Commands)**.**

$$\begin{aligned}
\mathbb{C} ::= \ & \mathtt{skip} \mid \mathtt{x} := \mathtt{E} \mid [\mathtt{E}_1] := \mathtt{E}_2 \mid \mathtt{x} := [\mathtt{E}] \mid \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3) \mid \mathtt{x} := \mathtt{FAS}(\mathtt{E}_1, \mathtt{E}_2) \mid \\
& \mathtt{x} := \mathtt{new}(\mathtt{E}) \mid \mathtt{dispose}(\mathtt{E}) \mid \mathbb{C}_1 \ ; \ \mathbb{C}_2 \mid \mathbb{C}_1 \ \|\ \mathbb{C}_2 \mid \mathtt{var}\ \mathtt{x} = \mathtt{E}\ \mathtt{in}\ \mathbb{C} \mid \\
& \mathtt{if}\ (\mathtt{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2 \mid \mathtt{while}\ (\mathtt{B})\ \mathbb{C} \mid \mathtt{let}\ f(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathbb{C}_1\ \mathtt{in}\ \mathbb{C}_2 \mid \mathtt{y} := f(\mathtt{E}_1, \ldots, \mathtt{E}_n)
\end{aligned}$$

For well-formedness of $\mathbb{C}_1 \ \| \ \mathbb{C}_2$, we require $mod(\mathbb{C}_1 \ \| \ \mathbb{C}_2) = \emptyset$. We intend to model shared memory concurrency through the heap without interference through shared variables, which this condition does not restrict. We also require the obvious restriction for $\mathtt{let}\ f(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathbb{C}_1\ \mathtt{in}\ \mathbb{C}_2$ that $pv(\mathbb{C}_1) \subseteq \{\mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{ret}\}$.

Definitions of $mod(\mathbb{C})$, $pv_{\mathtt{E}}(\mathtt{E})$, $pv_{\mathtt{B}}(\mathtt{B})$ and $pv_{\mathbb{C}}(\mathbb{C})$ are provided in the Appendix A. I will later omit the subscripts when unambiguous.

4.1.1 Trace semantics of commands

$$\begin{aligned}
\mathsf{Addr} &\triangleq \mathbb{N} \\
\sigma_p \in \mathsf{PStore} &\triangleq \mathsf{PVar} \rightharpoonup_f \mathsf{Val} \\
h \in \mathsf{Heap} &\triangleq \mathsf{Addr} \rightharpoonup_f \mathsf{Val}
\end{aligned}$$

We represent the state of a program execution with a program configuration. This is either a tuple of a program store, heap and command, or $\lightning$ (denotating a configuration which has faulted). A ✓ denotes a terminated thread.

$$c \in \mathrm{Conf} \triangleq (\mathsf{PStore} \times \mathsf{Heap} \times (\mathsf{Cmd} \uplus \checkmark)) \uplus \{\lightning\}$$

$$s \in \mathrm{Sched} \triangleq \{\mathrm{loc}, \mathrm{env}\}$$

Programs execute within a function implementation context, with components for their program variable parameters and the command to be executed.

$$\varphi \in \mathsf{FImpl} \triangleq \mathsf{FName} \rightharpoonup (\mathsf{PVar}^*, \mathsf{Cmd})$$

We reason about traces of commands with respect to a standard operational semantics given by a relation $\rightarrow_\varphi \subseteq \mathsf{Conf} \times \mathsf{Sched} \times \mathsf{Conf}$ defined in Appendix B. We write $a \xrightarrow{s}_\varphi b$ for $(a, s, b) \in \rightarrow_\varphi$, using an expression evaluation function $[\![\cdot]\!]_{\sigma_p} : (\mathsf{PExp} \uplus \mathsf{PBExp}) \rightarrow \mathsf{Val}$ (Appendix A). Given that the goal is to reason about resource safety and thread interference, we are assuming that expressions always evaluate and evaluate to an object of the right 'type' - type-checking and faults caused by such errors are unrelated to concurrency problems and out of the scope of this field. Therefore, we assume that in `while` (`B`) $\mathbb{C}$, it is always the case that $[\![\mathtt{B}]\!]_{\sigma_p} \in \mathsf{Bool}$, in $\mathtt{x} := [\mathtt{E}_1]$ that $[\![\mathtt{E}_1]\!]_{\sigma_p} \in \mathsf{Addr}$, etc.

Definition 4.1.2 (Traces)**.**

$$\tau \in \mathsf{Trace}_\varphi \triangleq \left\{ (c_0, s_0, c_1, s_1, ..., s_{n-1}, c_n) \;\middle|\; \forall i \in \mathbb{N}.\ c_i \xrightarrow{s_i}_\varphi c_{i+1} \right\}$$

$$\mathsf{Trace} \triangleq \bigcup_{\varphi \in \mathsf{FImpl}} \mathsf{Trace}_\varphi$$

We use standard indexing notation on traces, i.e. $\tau[0]$ will always refer to the first configuration c_0, and $\sigma_i, h_i, \mathbb{C}_i$ will always refer to the components of the i-th Conf in τ.

Finally, we define the semantics of commands:

Definition 4.1.3 (Trace Semantics of Commands)**.**

$$[\![\mathbb{C}]\!]_\varphi \triangleq \left\{ \tau \;\middle|\; \begin{array}{l} \tau \in \mathsf{Trace}_\varphi \wedge \tau[0] = (\sigma_0, \_, \mathbb{C}) \\ \wedge\ pv(\mathbb{C}) \subseteq dom(\sigma_0) \end{array} \right\}$$

A trace τ is considered a concrete representation of an execution of command $\mathbb{C}$ in function implementation context φ, if it begins starting with $\mathbb{C}$, each step is determined by the operational semantics $\rightarrow_\varphi$ and each variable free in the command has a definition in σ_0 (avoiding non-concurrency related faults). By imposing no restrictions on the initial heap, it follows that the traces accepted by $[\![\mathbb{C}]\!]_\varphi$ are those which represent genuine program executions of $\mathbb{C}$.

Remark 4.1.1. TaDALive uses a complex inductive data structure called a PState in the third component of program configurations in order to manage nested local variable declarations and usages. I sidestep this complexity by providing a new operational semantics rule for handling local variables. This greatly eases later inductive proofs.

Remark 4.1.2. In order for TaDALive's trace semantics of commands and specifications to be directly comparable, there is a further reduction from their program configurations to a more concrete trace with no representation of the commands. Then, a third type of trace is used in defining the semantics of specifications. In contrast, TaDA 2.0 only has one type of trace which suffices as the semantics of all its components.

Remark 4.1.3. The semantics of TaDALive's commands (and specifications) are all infinite traces. Terminating executions are modelled by an infinite suffix of environment steps, and closed systems by traces in which environment steps are all zero. This is necessary in order to reason about termination. With the aim of reducing the semantic model to only that which is necessary for safety, I have elected to use finite traces as the semantics for commands, which makes them easier to reason about. However, in order to get the correct safety semantics for non-terminating executions, the semantics of commands in TaDA 2.0 include what could be thought of as *trace prefixes*: for any given trace in $[\![\mathbb{C}]\!]_\varphi$, every prefix of that trace is also in $[\![\mathbb{C}]\!]_\varphi$. This allows us to ensure we are also providing safety guarantees for all possible finite executions of non-terminating programs.

Remark 4.1.4. As the satisfaction of a TaDALive specification depends on both the liveness and safety conditions, TaDALive requires a fairness condition on the scheduler of its traces, which in turn requires explicit identification and tracking of which thread took which step in the operational semantics. Therefore it does not provide safety guarantees of unfairly scheduled traces (because the liveness result would not hold in these cases). TaDA 2.0 has no dependence on the scheduling and provides safety guarantees for all program executions.

4.2 Assertions

Assertions require the following:

LVar, a set of logical variables disjoint from PVar
$\mathbf{t} \in \mathsf{RType}$, a set of region types
$r \in \mathsf{RId}$, a set of region identifiers
$\lambda \in \mathsf{Lvl} \triangleq \mathbb{N}$
$a \in \mathsf{AState}$ a set of abstract states. This can be abstract values, as well as sets or lists of these, for example.
$G \in \mathsf{Guard}$ a set of all guards in the proof system.
$\mathsf{AVal} \triangleq \mathsf{Val} \cup \mathsf{AState} \cup \mathsf{Guard} \cup \mathsf{RId}$
$E \in \mathsf{LExp} ::= \mathsf{Exp}(\mathsf{LVar} \uplus \mathsf{PVar}, \mathsf{AVal})$ logical numerical expressions
$B \in \mathsf{LBExp} ::= \mathsf{BExp}(\mathsf{LVar} \uplus \mathsf{PVar}, \mathsf{AVal})$ logical boolean expressions

Observe that $\mathsf{PExp} \subseteq \mathsf{LExp}$ and $\mathsf{PBExp} \subseteq \mathsf{LBExp}$, so we may freely use PExps and PBExps which occur in commands directly in our assertions and specifications. Analagous definitions of pv hold for logical expressions and assertions.

Definition 4.2.1 (Atomicity tracking components)**.**

$$d ::= \blacklozenge \quad | \quad \Diamond \quad | \quad (E_1, E_2)$$

Definition 4.2.2 (Assertions)**.** Assertions are defined by the following grammar:

$$P ::= B \quad | \quad \exists x.P \quad | \quad P \implies Q \quad | \quad P \wedge Q \quad | \quad P \vee Q \quad | \quad \mathsf{emp} \quad | \quad P \star Q \quad | \quad \circledast_{i \in S}(P(i)) \quad |$$

$$E_1 \mapsto E_2 \quad | \quad \mathbf{t}^{\lambda}_{r}(E) \quad | \quad r \Mapsto d \quad | \quad \lceil G(E_1, ..., E_n) \rceil_r$$

where $\mathbf{t} \in \mathsf{RType}$, $\lambda \in \mathsf{Lvl}$, $r \in \mathsf{RId} \cup \mathsf{LVar}$, $G \in \mathsf{Guard} \cup \mathsf{LVar}$

4.2.1 World semantics of assertions

Ghost state is intended to provide additional machinery to reason about properties unobservable in the program configuration. As it is used to represent the restrictions on interference, TaDA 2.0 provides a logical representation of assertions to encode these expectations. This will be worlds. A world represents a *thread-local* view of the program configuration - they represent only the resource owned or partially owned by a thread (or set of threads). This will allow world composition to correspond to the program state from the combined perspective of several threads. The term view is from [28], which introduces the meta-theory of abstract local states which compose to form part of the wider system.

Define $\sigma_l \in \mathsf{LStore} \triangleq \mathsf{LVar} \rightharpoonup_f \mathsf{AVal}$, and $\varsigma \in \mathsf{Store} \triangleq \mathsf{PVar} \uplus \mathsf{LVar} \rightharpoonup_f \mathsf{AVal}$ the disjoint union of pairs of LStores and PStores, (i.e. all PVar's map to Vals.) Then expression evaluation extends to $[\![\cdot]\!]_\varsigma : (\mathsf{LExp} \uplus \mathsf{LBExp}) \to \mathsf{AVal}$.

Definition 4.2.3 (Worlds)**.** For some $\mathcal{R} \subseteq \mathsf{RId}$, a world $w \in \mathtt{World}_{\mathcal{R}}$, is the tuple $w = (h, \rho, \gamma, \chi)$ where
- h is the local heap, representing resources exclusively owned by the thread
- $\rho \in \mathtt{RMap} \triangleq \mathsf{RId} \rightharpoonup_f (\mathsf{RType} \times \mathtt{Lvl} \times \mathsf{AState})$ determines the shared regions the thread knows about
- $\gamma \in \mathtt{GMap} \triangleq \mathsf{RId} \rightharpoonup_f \mathtt{Guard}$ is the locally owned guards
- $\chi \in \mathtt{AMap}_{\mathcal{R}} \triangleq \mathcal{R} \to \mathtt{ATrack}$ is the atomicity tracking component of regions for which we are in the midst of doing atomic actions.

$\mathcal{R}$ contains the region identifiers which have been *opened* (i.e. we are in the midst of tracking an atomic update to a shared region with this identifer, and temporarily consider the concrete interpretation of the region to be part of our semantics instead of the shared region component).
We impose the additional restrictions on worlds to be well-formed: $\mathtt{dom}(\rho) = \mathtt{dom}(\gamma) \supseteq \mathcal{R}$ and $\forall r \in \mathsf{RId},\ \rho(r) = (\mathbf{t}, \_, \_) \implies \gamma(r) \in \mathcal{G}_{\mathbf{t}}$. This represents the condition that every shared region a thread knows about must have its information accessible in the maps ρ and γ and that if ρ says a region identifier has particular region type then the guard $\chi(r)$ must be contained within the guard algebra specific to that region.

Remark 4.2.1. TaDA's worlds contain components for CAP's abstract predicates. TaDALive's worlds do not, but do contain additional components for ghost state relating to liveness properties.

In order to understand a program configuration which is the composition of several concurrent threads, we need to be able to compose worlds in a such a way which produces a view of the execution from the combined perspective of the threads. In order to compose worlds, we first need to understand how to compose each component.

Definition 4.2.4 (Guard Algebra)**.** A guard algebra is a partial commutative monoid $(\mathsf{Grd}, \bullet, \{0\})$ with $\mathsf{Grd} \subseteq \mathsf{Guard}$. Our proofs are parameterised by a function $\mathcal{G}(\mathbf{t}) = (\mathcal{G}_{\mathbf{t}}, \bullet_{\mathbf{t}}, 0_{\mathbf{t}})$ from a region type to an associated guard algebra, where $\mathcal{G}_{\mathbf{t}} \subseteq \mathtt{Guard}$, so each region type has its own definition of composition for the guards of $\mathcal{G}(\mathbf{t})$.

Definition 4.2.5 (Atomicity Tracking Algebra)**.** An atomicity tracking algebra is a partial commutative monoid

$$\mathtt{ATrack} \triangleq ((\mathsf{AState} \times \mathsf{AState}) \uplus \{\blacklozenge, \lozenge\}, \bullet, \mathtt{Emp}_{\blacklozenge})$$

where

$$\mathtt{Emp}_{\blacklozenge} \triangleq (\mathsf{AState} \times \mathsf{AState}) \cup \{\lozenge\}$$

and as in TaDA, the operation is defined by

$$\begin{aligned}
&\blacklozenge \bullet \lozenge = \lozenge \bullet \blacklozenge = \blacklozenge \\
&\lozenge \bullet \lozenge = \lozenge \\
&\forall x, y \in \mathrm{AState},\ (x, y) \bullet (x, y) = (x, y)
\end{aligned}$$

All other combinations are undefined.

Definition 4.2.6 (Disjointness)**.** For some partial commutative monoid $(X, \bullet, X')$, $x, y \in X$ are disjoint, written $x \# y$, if and only if $x \bullet y \neq \bot$

Definition 4.2.7 (World Composition)**.** For $w_1 = (h_1, \rho_1, \gamma_1, \chi_1)$ and $w_2 = (h_2, \rho_2, \gamma_2, \chi_2) \in \mathtt{World}_{\mathcal{R}}$,

$$\begin{aligned}
h_1 \bullet h_2 &= h_1 \uplus h_2 \\
\rho_1 \bullet \rho_2 &= \rho_1 \text{ if } \rho_1 = \rho_2 \text{ and undefined otherwise} \\
\gamma_1 \bullet_\rho \gamma_2 &= \lambda r \in \mathtt{dom}(\rho),\ \gamma_1(r) \bullet_{\mathbf{t}} \gamma_2(r) \\
&\qquad \text{if } \forall r' \in \mathtt{dom}(\rho),\ \rho(r') = (\mathbf{t'}, _, _) \wedge \gamma_1(r') \bullet_{\mathbf{t}'} \gamma_2(r') \neq \bot, \text{ and } \bot \text{ otherwise} \\
\chi_1 \bullet_{\mathcal{R}} \chi_2 &= \lambda r \in \mathcal{R},\ \chi_1(r) \bullet \chi_2(r) \\
&\qquad \text{if } \forall r \in \mathcal{R},\ \chi_1(r) \bullet \chi_2(r) \neq \bot
\end{aligned}$$

Then, $\forall w_1,\ w_2 \in \mathtt{World}_{\mathcal{R}},\ w_1 \bullet w_2 = (h_1 \bullet h_2, \rho_1 \bullet \rho_2, \gamma_1 \bullet_{\rho_1} \gamma_2, \chi_1 \bullet_{\mathcal{R}} \chi_2)$.

As h_1 and h_2 represent a thread's exclusively owned resources, these compose only if they are disjoint. ρ_1 and ρ_2 defines the region type, region level and abstract state of shared resources, which play important roles in ensuring interference is within the protocols defined for a given region. To ensure each thread is 'playing by the same rules', worlds only compose if they agree exactly on shared regions. The composition of guards and atomicity tracking resources is lifted directly from the operation on the underlying algebra, so we may use this operation to reason about disjoint worlds, i.e. if a world is such that $\gamma(r) = G$, for some G which does not compose with any non-unit guard of the region type's guard algebra, then we may be certain that the guards owned by composed worlds may only represent a unit guard for that region identifier. Similarly, the operation on atomicity tracking components forces threads to agree on the location and justification of the linearisation point of a thread.

This gives us a world algebra $(\mathtt{World}_{\mathcal{R}}, \bullet, \mathtt{Emp}_{\mathcal{R}})$, where

$$\mathtt{Emp}_{\mathcal{R}} \triangleq \left\{ (\emptyset, \rho, \gamma, \chi) \;\middle|\; \begin{array}{l} \forall r \in \mathtt{dom}(\rho),\ \rho(r) = (\mathbf{t}, \_, \_) \implies \gamma(r) = 0_{\mathbf{t}} \\ \forall r \in \mathcal{R},\ \chi(r) \in \mathtt{Emp}_{\blacklozenge} \end{array} \right\}$$

We consider worlds to be empty when they contain no fully owned resources, i.e. the guards and atomicity tracking components are all units of their respective algebras and the heap is empty. We allow any shared regions, as these are not fully locally owned resources, and world composition requires equality of the ρ components.

We can lift world composition to sets of worlds in the following manner:

$$\forall p_1, p_2 \in \mathcal{P}(\mathtt{World}_{\mathcal{R}}).\ p_1 * p_2 \triangleq \{ w_1 \bullet_{\mathcal{R}} w_2 \mid w_1 \in p_1 \wedge w_2 \in p_2 \wedge w_1 \# w_2 \} \tag{4.1}$$

$$
\begin{array}{lcl}
\varsigma, w \models_{\mathcal{R}} B & \iff & [\![B]\!]_\varsigma \wedge w \in \mathtt{Emp}_{\mathcal{R}} \\
\varsigma, w \models_{\mathcal{R}} \exists \mathtt{x}.P & \iff & \exists v \in \mathsf{AVal}.\ \varsigma[\mathtt{x} \mapsto v], w \models_{\mathcal{R}} P \\
\varsigma, w \models_{\mathcal{R}} P \implies Q & \iff & \forall w'. w \preceq_R w' \wedge \varsigma, w' \models_{\mathcal{R}} P \implies \varsigma, w' \models_{\mathcal{R}} Q \\
\varsigma, w \models_{\mathcal{R}} P \wedge Q & \iff & (\varsigma, w \models_{\mathcal{R}} P) \wedge (\varsigma, w \models_{\mathcal{R}} Q) \\
\varsigma, w \models_{\mathcal{R}} P \vee Q & \iff & (\varsigma, w \models_{\mathcal{R}} P) \vee (\varsigma, w \models_{\mathcal{R}} Q) \\
\varsigma, w \models_{\mathcal{R}} \mathsf{emp} & \iff & w \in \mathtt{Emp}_{\mathcal{R}} \\
\varsigma, w \models_{\mathcal{R}} P \star Q & \iff & \exists w_1, w_2.\ w = w_1 \bullet w_2 \wedge (\varsigma, w_1 \models_{\mathcal{R}} P) \wedge (\varsigma, w_2 \models_{\mathcal{R}} Q) \\
\varsigma, w \models_{\mathcal{R}} \circledast_{i \in S}(P(i)) & \iff & (S = \emptyset \implies h \in \mathtt{Emp}_{\mathcal{R}}) \wedge \forall i \in S, \exists w_i \in \mathtt{World}_{\mathcal{R}}. \\
 & & \quad \varsigma, w_i \models_{\mathcal{R}} P(i) \wedge w = (\bullet_{\mathcal{R}, i \in S}\ w_i) \\
\varsigma, w \models_{\mathcal{R}} E_1 \mapsto E_2 & \iff & w = (h, \rho, \gamma, \chi) \wedge h = [[\![E_1]\!]_\varsigma \mapsto [\![E_2]\!]_\varsigma] \wedge (\emptyset, \rho, \gamma, \chi) \in \mathtt{Emp}_{\mathcal{R}} \\
\varsigma, w \models_{\mathcal{R}} \mathbf{t}_r^\lambda(E) & \iff & w = (\_, \rho, \_, \_) \wedge \rho([\![r]\!]_\varsigma) = (\mathbf{t}, \lambda, [\![E]\!]_\varsigma) \wedge w \in \mathtt{Emp}_{\mathcal{R}} \\
\varsigma, w \models_{\mathcal{R}} r \Mapsto d & \iff & w = (h, \rho, \gamma, \chi) \wedge \chi([\![r]\!]_\varsigma) = d \wedge (h, \rho, \gamma, \chi[[\![r]\!]_\varsigma \mapsto \Diamond]) \in \mathtt{Emp}_{\mathcal{R}} \\
\varsigma, w \models_{\mathcal{R}} \lceil G(E_1, ..., E_n) \rceil_r & \iff & w = (h, \rho, \gamma, \chi) \wedge \gamma([\![r]\!]_\varsigma) = [\![G]\!]_\varsigma([\![E_1]\!]_\varsigma, ..., [\![E_n]\!]_\varsigma) \\
 & & \quad \wedge (h, \rho, \gamma[[\![r]\!]_\varsigma \mapsto 0], \chi) \in \mathtt{Emp}_{\mathcal{R}}
\end{array}
$$

Figure 4.1: World satisfaction relation

In order to provide a semantics for assertions which is flexible enough to allow composition any sensible frame, we need to allow the 'addition' of shared regions to the logical justification of the state, as there may be shared regions represented elsewhere in the entire state not observed locally. To do this, we consider the semantics of assertions to be a set of upwards-closed worlds with respect to adding shared regions. Define the ordering on $\mathtt{World}_{\mathcal{R}}$, $\preceq_{\mathcal{R}}$, to be the smallest reflexive, transitive relation such that:

$$(h, \rho, \gamma, \chi) \ \preceq_{\mathcal{R}} \ (h, \rho[r \mapsto (\mathbf{t}, \lambda, a)], \gamma[r \mapsto 0_{\mathbf{t}}], \chi) \quad \text{if } r \notin dom(\rho)$$

Then

$$\mathtt{World}_{\mathcal{R}}^{\uparrow} \triangleq \{\ p \subseteq \mathtt{World}_{\mathcal{R}} \mid \forall w, w' \in \mathtt{World}_{\mathcal{R}}, w \in p \wedge w \preceq_{\mathcal{R}} w' \implies w' \in p\ \}$$

We can check that this is sensible by observing that the set of empty worlds is indeed upwards closed, and so can represent the logical instrumentation of any concrete state satisfying assertions with no resources (such as `true`).

Lemma 4.1 $(\forall \mathcal{R} \in \mathcal{P}(\mathsf{RId}).\ \mathtt{Emp}_{\mathcal{R}} \in \mathtt{World}_{\mathcal{R}}^{\uparrow})$**.**

The logical state is represented by a store and world. The world satisfaction relation $\models_{\mathcal{R}} \subseteq (\mathsf{Store} \times \mathtt{World}_{\mathcal{R}}) \times \mathsf{Assert}$ defines when a logical state is a genuine representation of an assertion, according to Figure 4.1. It is parameterised by $\mathcal{R}$, the region identifiers of open regions in the world.

Definition 4.2.8 (World Semantics of Assertions)**.** The semantics of an assertion P is defined to be the upwards closed set of worlds according to the world satisfaction relation:

$$\mathcal{W}[\![P]\!]_{\mathcal{R}}^{\varsigma} \triangleq \{\ w \mid \varsigma, w \models_{\mathcal{R}} P\ \}$$

Lemma 4.2 $(\forall P \in \mathsf{Assert}, \mathcal{R} \in \mathcal{P}(\mathsf{RId}), \varsigma \in \mathsf{Store}.\ \mathcal{W}[\![P]\!]_{\mathcal{R}}^{\varsigma} \in \mathtt{World}_{\mathcal{R}}^{\uparrow})$**.**

4.2.2 Atomicity context, inteference and stability

To successfully verify programs employing linearisability reasoning for race-freedom, TaDA 2.0 needs to be able to verify that a program which is not a primitive atomic command is in fact logically atomic. In doing so, it uses most of the existing infrastructure to verify the safety of each component command in the usual sequential way, with the atomicity context and atomicity tracking resource responsible for ensuring that the abstract state of the shared resource is only updated once. Therefore, we need to be able to track which shared regions we are in the process of verifying an abstract atomic program: this is the role played by the atomicity context.

Definition 4.2.9 (Atomicity Context)**.** An atomicity context $\mathcal{A}$ is a partial function from RId to $\mathcal{P}(\mathsf{AState} \times \mathsf{AState})$, usually written $\mathcal{A}(r) = (X, T)$ where $X \subseteq \mathsf{AState}$ and $T \subseteq \mathsf{AState} \times \mathsf{AState}$. We provide the shorthand $\text{interf}(\mathcal{A}, r) \triangleq X$ and $\text{tr}(\mathcal{A}, r) \triangleq T$. For every $r \in dom(\mathcal{A})$, we require $X \subseteq \{\ x \mid (x, \_) \in \text{tr}(\mathcal{A}, r)\ \}$.

Notation 4.1.

$$\mathcal{W}[\![P]\!]^{\varsigma}_{\mathcal{A}} \triangleq \mathcal{W}[\![P]\!]^{\varsigma}_{dom(\mathcal{A})}$$

TaDA is parameterised by a region interference function $\mathcal{T}_{\mathbf{t}} : \mathcal{G}_{\mathbf{t}} \to \mathcal{P}(\mathsf{AState} \times \mathsf{AState})$, which given a region type $\mathbf{t} \in \mathsf{RType}$ and a guard in the associated guard algebra, gives the set of transitions allowed by this guard on the region. We require the relation to be reflexive, and monotone in the guards, that is, $G_1 \preceq G_2 \implies \mathcal{T}_{\mathbf{t}}(G_1) \subseteq \mathcal{T}_{\mathbf{t}}(G_1)$, where the ordering $\preceq$ on the guards is the natural one induced by the operation of the monoid. We use this to determine the rely and guarantee- a thread guarantees to limit its interference on shared resources to only actions permitted by the guards it owns, and relies on the environment doing the same. The partial nature of the guard monoid and definition of world composition means that other threads can only own guards which are disjoint to that owned by the local thread, which forms the basis of the rely.

To bake the rely-guarantee directly into the logical instrumentation of the state of an execution, we define a world rely relation with respect to the atomicity context, $\mathbf{R}_{\mathcal{A}} \subseteq \mathtt{World}_{\mathcal{R}} \times \mathtt{World}_{\mathcal{R}}$ as the smallest reflexive, transitive closure which satisfies **WR1** and **WR2**, of Figure 4.2.
WR1 admits transitions permitted by $\mathcal{T}_{\mathbf{t}}(G)$ when the guards seen by the world ($\gamma(r)$) do not preclude the environment of owning G. If the world is waiting for an atomic action to happen on r ($\chi(r) \in \{\blacklozenge, \lozenge\}$), then the transitions are further restricted to those which are permitted by the environment interference of the atomicity context ($\text{interf}(\mathcal{A})$).
WR2 admits transitions where the environment performs the linearisation point.

$$\frac{\gamma(r) \# G \qquad (a_1, a_2) \in \mathcal{T}_{\mathbf{t}}(G) \qquad \chi(r) \in \{\blacklozenge, \lozenge\} \implies a_2 \in \text{interf}(\mathcal{A}, r)}{(h, \rho[r \mapsto (\mathbf{t}, \lambda, a_1)], \gamma, \chi)\ \mathbf{R}_{\mathcal{A}}\ (h, \rho[r \mapsto (\mathbf{t}, \lambda, a_2)], \gamma, \chi)}\ \textbf{WR1}$$

$$\frac{(a_1, a_2) \in \text{tr}(\mathcal{A}, r)}{(h, \rho[r \mapsto (\mathbf{t}, \lambda, a_1)], \gamma, \chi[r \mapsto \lozenge])\ \mathbf{R}_{\mathcal{A}}\ (h, \rho[r \mapsto (\mathbf{t}, \lambda, a_2)], \gamma, \chi[r \mapsto (a_1, a_2)])}\ \textbf{WR2}$$

Figure 4.2: World Rely Relation

The correctness of rely-guarantee reasoning hinges on being able to ensure that our logical instrumentation of the concrete state is not violable by updates to the concrete state contained within the rely. This is referred to as stability.

Definition 4.2.10 (Stability)**.**

$$View_{\mathcal{A}} \triangleq \{\ p \in \mathtt{World}^{\uparrow}_{\mathcal{A}} \mid \forall w, w' \in \mathtt{World}_{\mathcal{A}},\ w \in p \wedge w\ \mathbf{R}_{\mathcal{A}}\ w' \implies w' \in p\ \}$$

Define the stability of assertions using the judgement

$$\mathcal{A} \models P \text{ stable} \triangleq \forall \varsigma \in \mathsf{Store}.\ \mathcal{W}[\![P]\!]^{\varsigma}_{\mathcal{A}} \in View_{\mathcal{A}}$$

and say P is $\mathcal{A}$-stable.

For $\mathcal{A}'$ an extension of $\mathcal{A}$, we can coerce elements $p \in View_{\mathcal{A}}$ to $p' \in View_{\mathcal{A}'}$ by extending the atomicity tracking components for each additional shared region in every possible way.

Lemma 4.3 (Stability is closed under composition)**.**

$$\forall p, q \in View_{\mathcal{A}}.\ p * q \in View_{\mathcal{A}}$$

Pure assertions, as well as $x \mapsto v$ and emp are stable under any atomicity context, as are guards. Stability is preserved by $*, \wedge, \vee, \exists$. The important cases to check are regions assertions and atomicity tracking components, for which the following rules are usually sufficient:

Lemma 4.4 (Sufficient conditions for stability)**.**

$$\frac{\forall x \in X, x' \in \mathsf{AState}, G' \in \mathcal{G}_{\mathbf{t}}.\ G' \# G(x) \wedge (x, x') \in \mathcal{T}_{\mathbf{t}}(G') \implies x' \in X}{\mathcal{A} \models \exists x \in X.\ \mathbf{t}_r^{\lambda}(x) * \lceil G(x) \rceil_r \text{ stable}}$$

$$\frac{\text{interf}(\mathcal{A}, r) = X}{\mathcal{A} \models \exists x \in X.\ \mathbf{t}_r^{\lambda} * r \Rightarrow \blacklozenge \text{ stable}} \qquad \frac{r \in dom(\mathcal{A})}{\mathcal{A} \models r \Rightarrow \Diamond \vee r \mapsto (\_, \_) \text{ stable}}$$

4.2.3 Concrete semantics of assertions

Up until this point, we have constructed logical representations of the concrete state, in order to use the language of assertions to specify commands and argue that our programs obey the restraints imposed by different types of ghost state. In order for these logical instrumentations to be meaningful, we need to understand when a logical instrumentation represents a program configuration. As the logical instrumentations are thread-local, this will involve composing arbitrary but disjoint frames (worlds representing the environment threads), and importing a representation for the shared resources.

The second is achieved by establishing region interpretations for shared regions which provide a logical representation of the underlying resources of the shared region as both assertions and as worlds.

Definition 4.2.11 (Region Interpretations)**.** The *syntactic* region interpretation $\mathcal{I}_r = (\mathbf{t}, l, a, P)$, where $fv(P) \subseteq \{r, l, a\}$ and $\emptyset \models P$ stable.
For each $\mathbf{t} \in \mathsf{RType}$, the *semantic* region interpretation is $\mathcal{I}_{\mathbf{t}}[\![\cdot]\!] : \mathsf{RId} \times \mathsf{Lvl} \times \mathsf{AState} \to View_{\emptyset}$

Observe that we require shared regions to be stable, i.e. their underlying concrete representation is inviolable by actions of the local or environment thread, and remember that being a $View_{\mathcal{A}}$ requires the semantic region interpretation to be an upwards-closed set of worlds, so this is a sensible semantics. Frequently, the definition of P may depend on r and l, so we consider these LVars.

Notation 4.2. Where $\mathcal{I}_r = (\mathbf{t}, l, a, P)$,

$$\mathcal{I}(\mathbf{t}_{\mathtt{E}_1}^{\lambda}(\mathtt{E}_2)) = P\,[\mathtt{E}_1/r, \lambda/l, \mathtt{E}_2/a]$$

For well-formedness, TaDA 2.0 requires that the semantic and syntactic definitions coincide:

$$\mathcal{I}_{\mathbf{t}}[\![r, \lambda, a]\!] = \mathcal{W}[\![\mathcal{I}(\mathbf{t}_r^{\lambda}(a))]\!]_{\emptyset}^{\emptyset}$$

These semantics can be coerced to stable sets of worlds of any atomicity context and thus have the flexibility needed to compose with other worlds.

Now that we have a concrete representation of the non-exclusively owned resources, we can finally determine how logical states correspond to concrete resources.

Definition 4.2.12 (Region Collapse)**.** For $\lambda \in \text{Lvl}$, $w \in \mathtt{World}_{\mathcal{A}}$, let

$$\text{closed}(\lambda, w) = \{\ r \in dom(\rho) \mid \rho(r) = (\_, \lambda', \_) \wedge \lambda' < \lambda\ \}$$

Define region collapse to be

$$w{\downarrow^{\lambda}} \triangleq \left\{ w \bullet w_1 \bullet ... \bullet w_n \;\middle|\; \begin{array}{l} \text{closed}(\lambda, w) = \{r_1, ..., r_n\} \wedge \rho_w(r_i) = (\mathbf{t}_i, \lambda_i, a_i) \\ \implies w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda_i, a_i]\!] \end{array} \right\}$$

where we implicitly coerce $w_i \in \mathtt{World}_{\emptyset}$ to $w_i \in \mathtt{World}_{\mathcal{A}}$ by extending the atomicity tracking component to be compatible with the existing ghost state.

Definition 4.2.13 (Reification)**.** The *world reification of w at level λ*, is defined as

$$\lfloor w \rfloor_\lambda \triangleq \{h \in \mathsf{Heap} \mid (h, \_, \_, \_) \in w{\downarrow^\lambda}\}$$

and for any $p \in \mathtt{World}^\uparrow_{\mathcal{A}}$, its *reification at level* λ is

$$[\![p]\!]_\lambda \triangleq \bigcup_{w \in p} \lfloor w \rfloor_\lambda$$

For any assertion, we have constructed the set of heaps satisfying it to be those corresponding to the heaps of the worlds of its semantics, composed with any additional shared resources. With the ultimate goal here being to define traces as the semantics of specifications, we understand when a heap can represent an assertion, and now need to understand when a transitions between concrete heaps respects the logical instrumentation imposed by the assertion's worlds.

4.2.4 Frame-preserving updates and generalised implication

The notion of transition for logical instrumentation is the *frame-preserving update*. Earlier, stability was defined with the intention of making the sets of worlds used as semantics for assertions resistant to permitted interference. Now, this definition is used to define precisely when an update to a heap can be represented by a change in logical instrumentation which does not violate its guarantee.

Definition 4.2.14 (Frame-preserving update)**.** For any $h_1, h_2 \in \mathsf{Heap}, p_1, p_2 \in \mathrm{World}^\uparrow_{\mathcal{A}}, \lambda \in \mathsf{Lvl}$, we define the frame-preserving update as follows

$$(h_1, h_2) \models_{\lambda,\mathcal{A}} p_1 \twoheadrightarrow p_2 \iff \forall f \in \mathrm{View}_{\mathcal{A}}.\ h_1 \in [\![p_1 * f]\!]_\lambda \implies h_2 \in [\![p_2 * f]\!]_\lambda$$

This says that an update from $[\![p_1]\!]_\lambda$ to $[\![p_2]\!]_\lambda$ is frame-preserving on heap h_1 if it updates it to h_2 without modifying any stable frames. By ensuring that environment steps are only made by frame-preserving updates, we get the essence of the rely, while ensuring that all local steps are frame-preserving gives us the essence of the guarantee.

Lemma 4.5 (Frame-preserving updates can be augmented with stable worlds)**.** For $\mathcal{A} \in \mathsf{AContext}$, $r, p, q \in \mathtt{World}^\uparrow_{\mathcal{A}}, h_0, h_1 \in \mathsf{Heap}$

$$\begin{array}{l}\textbf{if } \mathcal{A} \models r \text{ stable } \textbf{and } (h_0, h_1) \models_{\lambda,\mathcal{A}} p \twoheadrightarrow q \\ \textbf{then } (h_0, h_1) \models_{\lambda,\mathcal{A}} p * r \twoheadrightarrow q * r\end{array}$$

This machinery allows us to consider a generalised notion of logical implication, which exists in other concurrent separation logics, termed viewshift and denoted $\Rrightarrow$. It is strictly stronger than logical implication, and is used to define when the heap satisfying some logical instrumentation implies that the heap satisfies some other logical instrumentation - that one assertion can be viewshifted to another. This requires not just the heap to satisfy the new logical state, but for this transition to have no impact on any potential stable frames, and gives us the additional expressivity of updating ghost state such as guards and region assertions.

Definition 4.2.15 (Viewshift)**.** For $p_1, p_2 \in \mathtt{World}^\uparrow_{\mathcal{R}}$, we write $\lambda, \mathcal{A} \models p_1 \Rrightarrow p_2$ when

$$\forall h \in \mathsf{Heap}, (h, h) \models p_1 \twoheadrightarrow p_2$$

For assertions $P,\ Q$, let P viewshifts to Q, written $\lambda, \mathcal{A} \models P \Rrightarrow Q$, mean

$$\forall \varsigma \in \mathsf{Store},\ \lambda, \mathcal{A} \models \mathcal{W}[\![P]\!]^\varsigma_{\mathcal{A}} \Rrightarrow \mathcal{W}[\![Q]\!]^\varsigma_{\mathcal{A}}$$

An important use case of the viewshift is to give shared regions the property $\mathbf{t}^\lambda_r(a) \star Q \Rrightarrow Q$, so that they can be created from locally owned resources satisfying their implementation.

Remark 4.2.2. This method of handling the rely and guarantee has a symmetry in the way the definition is not dependent on which thread we consider (indeed, the guarantee, expressed by the frame-preserving update, can be though of as not violating the rely of any possible environment threads).

Chapter 5

TaDA 2.0 - Program Logic

5.1 Specifications

TaDA 2.0's specifications are a hybrid of the typical Hoare triples and TaDA's atomic triples, as in TaDA and TaDALive.

Definition 5.1.1 (Specifications)**.** Specifications, $\mathbb{S} \in \mathsf{Spec}$, have the general form

$$\mathbb{V}x \in X.\langle\ P_h \mid P_a(x)\ \rangle\ \cdot\ \exists y.\langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle_{\lambda,\mathcal{A}},$$

We call the P_h and Q_h assertions the Hoare pre- and post-condition, or private pre- and post-condition, in reference to their usage in verifying specifications using standard separation logic reasoning. The resources described by P_h and Q_h should be considered as locally owned, although this is slightly misleading as they frequently do contain shared regions which occur in sequential proofs. P_a and Q_a are the atomic pre- and postcondition, frequently thought of as shared resources, although again this is not quite correct as it may also describe exclusively locally owned resources. For the best intuition, P_a and Q_a should be thought of as resources which are subject to additional interference, with ownership over the resources only at the linearisation point.

Function specification contexts provide a specification for functions for which we have an implementation. The first component refers to the parameters of the variables, with PVar^* an ordered list of pairwise distinct program variables.

Definition 5.1.2 (Function Specification Contexts)**.**

$$\Phi \in \mathsf{FSpec} \triangleq \mathsf{FName} \rightharpoonup (\mathsf{PVar}^*, \mathbb{S})$$

For well-formedness, require $pv(P) \subseteq \vec{\mathtt{x}}$ and $pv(Q) = \{\mathtt{ret}\}$.

I provide syntactic sugar for the more traditional Hoare and atomic triples, and emphasise that these are not *different* triples, but simply syntactic sugar for hybrid triples. As such, they allow the application of proof rules without explicit conversion:

Notation 5.1 (Hoare triples)**.**

$$\Phi, \lambda, \mathcal{A} \vdash \{P\}\ \mathbb{C}\ \{Q\} \triangleq\ \Phi, , \lambda, \mathcal{A} \vdash \mathbb{V}x \in \mathsf{AVal}.\langle\ P \mid \mathsf{emp}\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q \mid \mathsf{emp}\ \rangle$$

Notation 5.2 (Atomic triples)**.**

$$\begin{aligned}\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle P(x)\rangle\ \mathbb{C}\ \exists y.\langle Q(x,y)\rangle \triangleq \\ \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\left\langle \begin{array}{l} \vec{\mathtt{v}} = \vec{v} \star \vec{\mathtt{v}'} = \vec{v'} \star \\ \vec{\mathtt{v}}_0 = \vec{v_0} \star \vec{\mathtt{v}}_1 = \vec{v_1} \end{array} \middle|\ P(x)[\vec{v}/\vec{\mathtt{v}}, \vec{v'}/\vec{\mathtt{v}'}, \vec{v_0}/\vec{\mathtt{v}_0}] \right\rangle\ \mathbb{C} \\ \exists y.\left\langle\ \vec{\mathtt{v}} = \vec{v} \star \vec{\mathtt{v}}_0 = \vec{v_0} \star \vec{\mathtt{v}}_1 = \vec{v_1} \middle|\ Q(x,y)[\vec{v}/\vec{\mathtt{v}}, \vec{v'}/\vec{\mathtt{v}'}, \vec{v_1}/\vec{\mathtt{v}_1}]\ \right\rangle\end{aligned}$$

where $\vec{\mathtt{v}} = pv(\mathbb{C}) \setminus mod(\mathbb{C})$, $\vec{\mathtt{v}'} = mod(\mathbb{C})$, $\vec{\mathtt{v}}_0 = pv(P(x)) \setminus pv(\mathbb{C})$ and $\vec{\mathtt{v}}_1 = pv(Q(x,y)) \setminus pv(\mathbb{C})$.

The requirement of the atomic pre- and postcondition to not contain any program variables makes it crucial to get this transformation right. In particular, the details of the soundness proof require that no new logical variables are introduced in the postcondition, and of course, the usual requirement that the value of logical variables cannot change during program execution. According to these requirements, we ensure that every program variable referred to in the command, pre- and post-condition appear in the Hoare pre-condition, and use the existential y to refer to the updated value of modified program variables in hybrid specifications.

Remark 5.1.1. The transformation for atomic triples in the TaDALive paper is incorrect, because it does not ensure that all logical variables (except for the existentially quantified y) used in the postcondition exist in the precondition. This is a technical detail but necessary for soundness. Indeed, TaDALive makes reference to the complications caused by atomically modified program variables, and says

> "The program variables mentioned in the atomic pre-/post-conditions refer to the value stored in them *at the beginning* of the execution of the command. Most commonly variables used this way are not modified by the command."

This is already a strange notational convention, but the second assumption is not true and actually means their proof rules for read, CAS and FAS are not well-defined. I will discuss this in detail in the next section.

We may omit the $\exists y$ in cases y does not occur in the postcondition, which is frequently, as it is only required when the state at the linearisation point is nondeterministic and the postcondition depends on this non-deterministic choice. We may omit the environment liveness assumption $\mathbb{V}x \in X$ in the case no environment interference is allowed, i.e. $x \notin fv(P_a) \cup fv(Q_h) \cup fv(Q_a)$. Its absence refers to the assumption $\forall x \in \mathsf{AVal}$, as in the notation for Hoare triples, but has uses in hybrid and atomic triples as well.

5.1.1 Trace semantics of a specification

We have constructed sets of heaps as the concrete semantics for assertions, as well as worlds as the logical instrumentation of the concrete state and frame-preserving updates as the transitions which respect ghost states. We now construct the trace semantics for specifications of traces which begin in a program configuration satisfying the precondition and end in one satisfying the postcondition. Let $\mathsf{AVal}' \triangleq (\mathsf{AVal} \uplus (\mathsf{AVal} \times \mathsf{AVal}))$. Then define specification states as follows.

Definition 5.1.3 (Specification States)**.**

$$(p_h, p_a, v) \in \mathsf{SState} \triangleq \ (\mathrm{View}_{\mathcal{A}} \times (\mathsf{AVal}' \to \mathtt{World}_{\mathcal{A}}^{\uparrow}) \times \mathsf{AVal}')$$

We consider an SState to be all the possible logical interpretations of the concrete state, where the p_h component should be thought of as representing the locally owned resources (described by the Hoare precondition), the p_a component should be thought of as a function from the abstract state dependent on the environment interference assumption to the shared resources (described by the atomic precondition) and v the abstract state used by the environment interference assumption.
We use these to check that a trace is safe - that either it does not fault, or if it does, it was the environment to blame, where blame is assigned by examining whether the transitions of program configurations represent a transition of logical instrumentations which obeys the assumptions (on stability and environment interference).

The trace safety judgement is how we determine whether a trace indeed satisfies the safety assumptions on it. This is more than just whether local steps fault, and requires examining the possible logical instrumentations of the concrete state to assign *blame* for potential faults. We only need to find one such logical instrumentation of the local steps of a trace to justify that the local step did not fault and stayed within its guarantee, but the local trace needs to be able to accommodate for all possible logical instrumentations of environment steps. This duality inspires the trace safety definition from alternating automata, by which we accept a local step if we can find some frame-preserving update which justifies it, but branch for all possible frame-preserving updates representing environment steps. This is seen in particular by the construction of S in Env. Observe also by the implications in the Env and Env' rules that if there are no possible safe instrumentations of the environment step then the semantics are 'shortcircuited', i.e. we have assigned the blame to the environment and S will be empty (representing that there are no safely reachable end states). This is also the case in Env$\lightning$. In contrast, failure to find

$$\frac{}{(\sigma, h, \mathbb{C}) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(p_h, p_a, v)\}}\ \textsc{Term}$$

$$\frac{\begin{array}{c}(\sigma_2, h_2, \mathbb{C}_2)\tau \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S \qquad (h_1, h_2) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * p_a(v) \\ \mathbb{C}_2 = \checkmark \implies v \in \mathsf{AVal} \times \mathsf{AVal} \wedge p'_h = \mathcal{W}[\![Q_h(v)]\!]^{\sigma_l \circ \sigma_2}_{\mathcal{A}}\end{array}}{(\sigma_1, h_1, \mathbb{C}_1)\ \mathrm{loc}\ (\sigma_2, h_2, \mathbb{C}_2)\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S}\ \textsc{Stutter}$$

$$\frac{\begin{array}{c}(\sigma_2, h_2, \mathbb{C}_2)\tau \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, \langle v, v'\rangle), S \\ (h_1, h_2) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * \mathcal{W}[\![Q_a(v', v'')]\!]^{\sigma_l}_{\mathcal{A}} \qquad \mathbb{C}_2 = \checkmark \implies p'_h = \mathcal{W}[\![Q_h(v, v')]\!]^{\sigma_l \circ \sigma_2}_{\mathcal{A}}\end{array}}{(\sigma_1, h_1, \mathbb{C}_1)\ \mathrm{loc}\ (\sigma_2, h_2, \mathbb{C}_2)\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S}\ \textsc{LinPt}$$

$$\frac{\begin{array}{c}S = \bigcup_{v' \in X.\ E(v')} S_{v'} \qquad v \in \mathsf{AVal} \qquad \forall v \in X.\ E(v') \implies (\sigma, h_2, \mathbb{C})\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S_{v'} \\ E(v') \triangleq \exists p_e, p'_e.\ h_1 \in [\![p_h * p_a(v) * p_e]\!]_\lambda \wedge (h_1, h_2) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p'_e\end{array}}{(\sigma, h_1, \mathbb{C})\ \mathrm{env}\ (\sigma, h_2, \mathbb{C})\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S}\ \textsc{Env}$$

$$\frac{\text{if } \exists p_e, p'_e.\ h_1 \in [\![p_h * p_e]\!]_\lambda \wedge (h_1, h_2) \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e \text{ then } (\sigma, h_2, \mathbb{C})\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, \langle v, v'\rangle), S \text{ else } S = \emptyset}{(\sigma, h_1, \mathbb{C})\ \mathrm{env}\ (\sigma, h_2, \mathbb{C})\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, \langle v, v'\rangle), S}\ \textsc{Env'}$$

$$\frac{}{(\sigma, h, \mathbb{C})\ \mathrm{env}\ \lightning \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \emptyset}\ \textsc{Env}\lightning$$

Figure 5.1: Trace safety judgment

a frame-preserving update representing the update for local steps results in the trace safety judgement rejected the trace.

Definition 5.1.4 (Trace Safety)**.** Let $\mathbb{S} \in \mathsf{Spec}$ with components as in (5.1.1). The *trace safety judgement* is a relation $\models_{\mathbb{S}} \subseteq \mathsf{Trace} \times (\mathsf{LStore} \times \mathrm{SState} \times \mathcal{P}(\mathrm{SState}))$, with elements denoted as $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$, defined by the rules in Figure 5.1.

Remark 5.1.2. In order to collect liveness related ghost state and later check the trace fulfills its obligations, TaDALive defines specification traces, with states (σ, h, p_h, p_a, v) and transitions determined by the transitions of logical instrumentations which are deemed safe. Specification traces are constructed by the trace safety judgement, and then world traces for liveness from those. Because this separation between checking safety and liveness properties is built into the TaDALive semantic model, it is easier to extract only the necessary constructions for safety properties here. So the semantic model of TaDA 2.0 only has one type of trace (compared to TaDALive's four) sufficing for the semantics of each component.

Remark 5.1.3. The final component of the trace safety judgement intuitively corresponds to sets of reachably safe end states, where reachably safe refers to logical instrumentations of the final concrete state where each transition made to get there is safe. There are very little requirements on this component, and so it may be possible in future to explore removing it, further simplifying the trace safety judgement. I require it for the soundness proof in order to be able to reason about the postcondition when appending steps to a trace.

Definition 5.1.5 (Specification semantics)**.**

$$[\![\mathbb{S}]\!] \triangleq \left\{ (\sigma_0, h_0, \mathbb{C}_0)\tau \;\middle|\; \begin{array}{r}(\sigma_0, h_0, \mathbb{C}_0)\tau \in \mathsf{Trace},\ \forall \sigma_l \in \mathsf{LStore}, v \in X.\ \textbf{if } h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda \\ \textbf{then } \exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S\end{array} \right\}$$

where

$$p_h = \mathcal{W}[\![P_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$$
$$p_a = \lambda x.\ \mathcal{W}[\![P_a(x) * x \in X]\!]_{\mathcal{A}}^{\sigma_l}$$

A trace satisfies a specification if for any additional logical instrumentation (the LStore and abstract state $v \in X$) such that the first program configuration is a concrete representation of the precondition, then the trace is safe (which encompasses the postcondition).

Remark 5.1.4. TaDALive does not have any notion of LStore in its trace semantics of specifications, and thus the trace safety judgement is not strictly well-defined where the postcondition is checked. I resolve this by quantifying over logical stores and providing them as a component of the trace safety judgement which is constant across the entire trace, modelling the unchanging value of logical variables. The universal quantification is necessary to obtain the correct semantics upon satisfaction of the precondition.

Lemma 5.1 (Specifications of skip)**.** Let $\mathbb{S} \in \mathsf{Spec}$ and $\Phi \in \mathsf{FSpec}$. The following holds:

if $\models_\Phi \texttt{skip} : \mathbb{S}$
then $\forall v \in X, \exists v' \in \mathsf{AVal}.\ \lambda, \mathcal{A} \models P_h \star P_a(v) \Rrightarrow Q_h(v, v') \star Q_a(v, v')$.

Remark 5.1.5. The ✓ is simply a technical feature to ensure that all programs can take a step - otherwise its trace semantics would only be single states, and we could not verify the postcondition. This would cause unsoundness in sequencing `skip` with other commands.

Definition 5.1.6 (Semantic Judgement)**.** Our semantic judgement is that a command satisfies a specification with a given function implementation when:

$$\models_\varphi \mathbb{C} : \mathbb{S} \iff [\![\mathbb{C}]\!]_\varphi \subseteq [\![\mathbb{S}]\!]$$

Definition 5.1.7 (Semantics of function implementations)**.** We say that a function implementation φ satisfies a function specification context Φ, and denote it as $\models \varphi : \Phi$, when

$$\forall f, \vec{\mathtt{x}}, \mathbb{S}.\ \Phi(f) = (\vec{\mathtt{x}}, \mathbb{S}) \implies \exists \mathbb{C}.\ \varphi(f) = (\vec{\mathtt{x}}, \mathbb{C}) \wedge \models_\varphi \mathbb{C} : \mathbb{S}$$

Definition 5.1.8 (Generalised Semantic Judgement)**.** The generalised semantic judgement, that a command satisfies a specification in a given function specification context Φ is

$$\models_\Phi \mathbb{C} : \mathbb{S} \iff \forall \varphi.\ \models \varphi : \Phi \implies \models_\varphi \mathbb{C} : \mathbb{S}$$

5.2 Proof Rules

Definition 5.2.1 (Syntactic Judgement)**.** The generalised syntactic judgement is $\vdash_\Phi \mathbb{C} : \mathbb{S}$, written

$$\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X. \langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y. \langle\ Q_h(x, y) \mid Q_a(x, y)\ \rangle$$

with a well-formedness conditions that the Hoare pre- and postconditions P_h and $P_a(x)$ are $\mathcal{A}$-stable, $\mathtt{pv}(P_a(x)) = \mathtt{pv}(Q_a(x, y)) = \emptyset$, $\exists x \in X. P_a(x)$ is $\mathcal{A}$-stable and $X \subseteq \mathsf{AVal}$. These are parameterised by a level λ, atomicity context $\mathcal{A}$ and function specification context Φ. Its definition is given by the proof rules in Figures 5.2 to 5.5.

The TaDA 2.0 proof system has rules for each command construct, as well as a number of standard logical rules, some new logical rules to assist in atomicity rules, and three atomicity rules. ASSIGN, MUTATE, ALLOC and DEALLOC are as in TaDALive. READ, CAS and FAS are completely new and have replaced the atomic triples with hybrid triples explicitly handling the mutated program variable. The TaDALive READ rule was

$$\frac{}{\lambda, \mathcal{A} \vdash_\Phi \mathbb{V}v.\ \langle \mathtt{E} \mapsto v \rangle \mathtt{x} := [\mathtt{E}]\ \langle\ \mathtt{E} \mapsto v \wedge \mathtt{x} = v\ \rangle}$$

Unfolding this as per their syntactic sugar gives

$$\frac{}{\begin{array}{ll} \lambda, \mathcal{A} \vdash_\Phi \mathbb{V}v. & \langle\ \vec{v_0} = fv(\mathtt{E}) \mid \mathtt{E}[\vec{v_0}/\mathtt{fv}(()\mathtt{E})] \mapsto v\ \rangle \\ & \mathtt{x} := [\mathtt{E}] \\ \exists y.\ \langle\ \vec{v_0} = fv(\mathtt{E}) \star x = \mathtt{x} \mid \mathtt{E}[\vec{v_0}/fv(\mathtt{E})] \mapsto v \wedge x = v\ \rangle & \end{array}}$$

$$\textbf{(Assign)}\ \frac{}{\Phi, \lambda, \mathcal{A} \vdash \{\mathtt{x} = v\}\ \mathtt{x} := \mathtt{E}\ \{\mathtt{x} = \mathtt{E}[v/\mathtt{x}]\}}$$

$$\textbf{(Read)}\ \frac{\vec{\mathtt{w}} = fv(\mathtt{E}) \setminus \{\mathtt{x}\}}{\Phi, \lambda, \mathcal{A} \vdash \forall n \in \mathbb{Z}.\left\langle \begin{array}{l|l} \mathtt{x} = x \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \end{array} \right\rangle\ \mathtt{x} := [\mathtt{E}]\ \left\langle \begin{array}{l|l} \mathtt{x} = n \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \end{array} \right\rangle}$$

$$\textbf{(Mutate)}\ \frac{}{\Phi, \lambda, \mathcal{A} \vdash \forall n \in \mathbb{Z}.\langle \mathtt{E}_1 \mapsto n \rangle\ [\mathtt{E}_1] := \mathtt{E}_2\ \langle\ \mathtt{E}_1 \mapsto \mathtt{E}_2\ \rangle}$$

$$\textbf{(CAS)}\ \frac{\vec{\mathtt{w}} = fv(\mathtt{E}_1) \cup fv(\mathtt{E}_2) \cup fv(\mathtt{E}_3) \setminus \{\mathtt{x}\}}{\begin{array}{l} \Phi, \lambda, \mathcal{A} \vdash \quad \forall n \in \mathbb{Z}.\left\langle \begin{array}{l|l} \mathtt{x} = x \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \end{array} \right\rangle \\ \qquad\qquad \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3) \\ \exists y.\left\langle \begin{array}{l|l} \mathtt{x} = y \star & (n = \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \wedge y = 1 \star \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto \mathtt{E}_3[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]) \vee \\ \vec{\mathtt{w}} = \vec{w} & (n \neq \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \wedge y = 0 \star \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n) \end{array} \right\rangle \end{array}}$$

$$\textbf{(FAS)}\ \frac{\vec{\mathtt{w}} = fv(\mathtt{E}_1) \cup fv(\mathtt{E}_2) \setminus \{\mathtt{x}\}}{\begin{array}{l} \Phi, \lambda, \mathcal{A} \vdash \forall n \in \mathbb{Z}.\left\langle \begin{array}{l|l} \mathtt{x} = x \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \end{array} \right\rangle \\ \qquad\qquad \mathtt{x} := \mathtt{FAS}(\mathtt{E}_1, \mathtt{E}_2) \\ \left\langle \begin{array}{l|l} \mathtt{x} = n \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \end{array} \right\rangle \end{array}}$$

$$\textbf{(Alloc)}\ \frac{}{\Phi, \lambda, \mathcal{A} \vdash \{\ \mathtt{E} \geq 0 \star \mathtt{x} = x\}\ \mathtt{x} := \mathtt{new}(\mathtt{E})\ \{\ \exists y, \vec{v}.\mathtt{x} = y \star \circledast_{0 \leq i \leq \mathtt{E}[x/\mathtt{x}]-1} (y + i \mapsto v_i)\ \}}$$

$$\textbf{(Dealloc)}\ \frac{}{\Phi, \lambda, \mathcal{A} \vdash \{\exists v.\ \mathtt{E} \mapsto v\}\ \mathtt{dispose}(\mathtt{E})\ \{\ \mathtt{emp}\ \}}$$

Figure 5.2: Proof rules: primitive commands

This is wrong for two reasons. The first is that a new logical variable is introduced in the postcondition, x, which for technical reasons is impossible - logical stores must remain constant across a program execution. Secondly, x represents the value of the program variable $\mathtt{x}$ after the execution of the command. Therefore it's obvious that the statements made in TaDALive about program variables representing their value from before the command in the postcondition is inconsistent with their proof rules. I partially resolve the issue of not being able to introduce new logical variables by ensuring a corresponding logical variable is provided by every program variable required by the pre- and post-condition, *in the precondition*. The unhandled case with this solution is when an atomically modified program variable is present, like the $\mathtt{x}$ in a READ, CAS or FAS instruction. As the value of $\mathtt{x}$ at the linearisation point is unknown in the initial state, we cannot bind a logical variable to carry this value forward into the postcondition, and must instead make deliberate use of the $\exists y$ to obtain this value. In the case of READ and FAS, it is sufficient to use the pseudoquantified variable in the environment interference assumption to bind the value of $\mathtt{x}$ in the postcondition, but the nondeterministic nature of the result of a CAS requires y. The TaDA 2.0 proof rule for read defined in Figure 5.2 is

$$\textbf{(Read)}\ \frac{\vec{\mathtt{w}} = fv(\mathtt{E}) \setminus \{\mathtt{x}\}}{\Phi, \lambda, \mathcal{A} \vdash \forall n \in \mathbb{Z}.\left\langle \begin{array}{l|l} \mathtt{x} = x \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \end{array} \right\rangle\ \mathtt{x} := [\mathtt{E}]\ \left\langle \begin{array}{l|l} \mathtt{x} = n \star & \\ \vec{\mathtt{w}} = \vec{w} & \mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \end{array} \right\rangle}$$

Observe that the definition explicitly omits logical variables which corresponded to earlier values of modified program variables from the postcondition.

It's important to note that this transformation results in a proof rule which has slightly unexpected semantics: the $\mathtt{x} = n$ component of the postcondition is necessarily in the Hoare part, which has the consequence of disconnecting this assignment from the atomic action, meaning that this specification for read asserts that:

- the value in the heap at $[\![\mathtt{E}]\!]_{\sigma_0}$ is retrieved atomically
- the program variable $\mathtt{x}$ is set to the value which was retrieved, *but not necessarily at the same time*

$$\textbf{(Seq)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}_1\ \{\ R\ \} \qquad \Phi, \lambda, \mathcal{A} \vdash \{\ R\ \}\ \mathbb{C}_2\ \{\ Q\ \}}{\Phi, \lambda, \mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}_1\ ;\ \mathbb{C}_2\ \{\ Q\ \}}$$

$$\textbf{(If)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \{\ P * \mathtt{B}\ \}\ \mathbb{C}_1\ \{\ Q\ \} \qquad \Phi, \lambda, \mathcal{A} \vdash \{\ P * \neg\mathtt{B}\ \}\ \mathbb{C}_2\ \{\ Q\ \}}{\Phi, \lambda, \mathcal{A} \vdash \{\ P\ \}\ \mathtt{if}\ (\mathtt{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2\ \{\ Q\ \}}$$

$$\textbf{(While)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \{\ P * \mathtt{B}\ \}\ \mathbb{C}\ \{\ P\ \}}{\Phi, \lambda, \mathcal{A} \vdash \{\ P\ \}\ \mathtt{while}\ (\mathtt{B})\ \mathbb{C}\ \{\ P * \neg\mathtt{B}\ \}}$$

$$\textbf{(Var)}\ \frac{\begin{array}{c}\mathtt{x} \notin fv(P_h) \cup fv(Q_h) \cup fv(\mathtt{E}) \\ \Phi, \lambda, \mathcal{A} \vdash \forall y \in X.\ \langle\ P_h \star \mathtt{x} = \mathtt{E} \mid P_a(y)\ \rangle\ \mathbb{C}\ \exists z.\langle\ Q_h(y,z) \mid Q_a(y,z)\ \rangle\end{array}}{\Phi, \lambda, \mathcal{A} \vdash \forall y \in X.\ \langle\ P_h \mid P_a(y)\ \rangle\ \mathtt{var}\ \mathtt{x} = \mathtt{E}\ \mathtt{in}\ \mathbb{C}\ \exists z.\langle\ Q_h(y,z) \mid Q_a(y,z)\ \rangle}$$

$$\textbf{(Par)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \{\ P_1\ \}\ \mathbb{C}_1\ \{\ Q_1\ \} \qquad \Phi, \lambda, \mathcal{A} \vdash \{\ P_2\ \}\ \mathbb{C}_2\ \{\ Q_2\ \}}{\Phi, \lambda, \mathcal{A} \vdash \{\ P_1 \star P_2\ \}\ \mathbb{C}_1\ \ ||\ \ \mathbb{C}_2\ \{\ Q_1 \star Q_2\ \}}$$

$$\textbf{(Let)}\ \frac{pv(\mathbb{S}_1) \subseteq \vec{\mathtt{x}} \cup \{\mathtt{ret}\} \qquad f \notin dom(\Phi) \qquad \vdash_\Phi \mathbb{C}_1 : \mathbb{S}_1 \qquad \Phi' = \Phi[f \mapsto (\vec{\mathtt{x}}, \mathbb{S}_1)] \qquad \vdash_{\Phi'} \mathbb{C}_2 : \mathbb{S}_2}{\vdash_\Phi \mathtt{let}\ f(\vec{\mathtt{x}}) = \mathbb{C}_1\ \mathtt{in}\ \mathbb{C}_2 : \mathbb{S}_2}$$

$$\textbf{(Call)}\ \frac{(\vec{\mathtt{x}}, \forall y \in Y.\langle\ P_h \mid P_a(y)\ \rangle\ \cdot \exists z.\langle\ Q_h(y,z,\mathtt{ret}) \mid Q_a(y,z)\ \rangle_{\lambda,\mathcal{A}}\ ) \in \Phi(f)}{\Phi, \lambda, \mathcal{A} \vdash \forall y \in Y.\langle\ P_h[\vec{\mathtt{E}}/\vec{x}] \mid P_a(y)\ \rangle\ \mathtt{a} := f(\vec{\mathtt{E}})\ \exists z.\langle\ Q_h(y,z,a) \mid Q_a(y,z)\rangle}$$

Figure 5.3: Proof rules: Hoare commands

as the retrieval
This is exemplified by an example program similar to that discussed in Section 3.1, which also satisfies the specification of the READ rule.

$$\mathtt{y} := [\mathtt{E}]\ ;\ \mathtt{x} := \mathtt{y}$$

This change in semantics is particularly interesting as although it is not quite a true reflection of the operational semantics, it is actually a more accurate model of how programs execute on hardware, where the memory retrieval operation is at a different level of abstraction to the setting of a program variable and although considered to be atomic by the programmer, is not technically atomic on the machine (although interference cannot cause any issue).
I propose that this change in semantics is potentially a better model for atomic operations such as read, CAS and FAS, as it closer reflects the physical machine and although it is not necessitated by the operational semantics it is still a correct specification, and potentially may open up further work into verification of programs executed in machines with a different memory model.

Remark 5.2.1. TaDA does not impose any requirements on the program variables of atomic pre- and postconditions, so in this respect TaDA 2.0 inherits slightly weaker proof rules than TaDA from TaDALive.

$$\textbf{(Frame)}\ \frac{mod(\mathbb{C}) \cap pv(R_h) = \emptyset \qquad mod(\mathbb{C}) \cap pv(R_a(x)) = \emptyset \qquad \mathcal{A} \models R_h \text{ stable} \qquad \forall x \in X.\ \mathcal{A} \models\ R_a(x) \text{ stable} \qquad \Phi, \mathcal{A}, \lambda \vdash \mathbb{V}x \in X.\langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}{\Phi, \mathcal{A}, \lambda \vdash \mathbb{V}x \in X.\langle\ P_h \star R_h \mid P_a(x) \star R_a(x)\ \rangle\ \mathbb{C}\ \exists y.\ \langle\ Q_h(x,y) \star R_h \mid Q_a(x,y) \star R_a(x)\ \rangle}$$

$$\textbf{(}\exists\textbf{Elim)}\ \frac{\forall x \in X.\ \Phi, \lambda, \mathcal{A} \vdash \{\ P(x)\ \}\ \mathbb{C}\ \{\ Q\ \}}{\Phi, \lambda, \mathcal{A} \vdash \{\ \exists x \in X. P(x)\ \}\ \mathbb{C}\ \{\ Q\ \}}$$

$$\textbf{(A}\exists\textbf{Elim)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}(x,z) \in X \times Z.\langle\ P_h \mid P_a(x,z)\ \rangle\ \mathbb{C}\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y,z)\ \rangle}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid \exists z \in Z. P_a(x,z)\ \rangle\ \mathbb{C}\ \exists y.\ \langle\ Q_h(x,y) \mid \exists z \in Z. Q_a(x,y,z)\ \rangle}$$

$$\textbf{(Lvl-weakening)}\ \frac{\lambda_1 \leq \lambda_2 \qquad \lambda_1, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}{\lambda_2, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}$$

$$\textbf{(Atomic-weaken)}\ \frac{\mathcal{A} \models P_h \star P \text{ stable} \qquad \forall x \in X, \mathcal{A} \models Q(x,y) \text{ stable} \qquad \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P \star P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\ \langle\ Q_h(x,y) \mid Q(x,y) \star Q_a(x,y)\ \rangle}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \star P \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \star Q(x,y) \mid Q_a(x,y)\ \rangle}$$

$$\textbf{(Cons)}\ \frac{\begin{array}{c}\mathcal{A} \models P_h \text{ stable} \\ \lambda, \mathcal{A} \models P_h \Rrightarrow P'_h \qquad \forall x \in X.\ \mathcal{A} \models P'_a(x) \Longleftrightarrow P_a(x) \qquad \forall (x,y) \in X \times Y.\ \mathcal{A} \models Q_h(x,y) \text{ stable} \\ \forall (x,y) \in X \times Y.\ \lambda, \mathcal{A} \models Q'_a(x,y) \Rrightarrow Q_a(x,y) \qquad \forall (x,y) \in X \times Y.\ \lambda, \mathcal{A} \models Q'_h(x,y) \Rrightarrow Q_h(x,y) \\ \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P'_h \mid P'_a(x)\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q'_h(x,y) \mid Q'_a(x,y)\ \rangle\end{array}}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}$$

$$\textbf{(Subst)}\ \frac{\begin{array}{c}f : X \to Y \qquad \forall x \in X.\ \mathcal{A} \models P'_a(x) \Longleftrightarrow P_a(f(x)) \\ \forall x \in X, z.\ \mathcal{A} \models Q_h(f(x),z) \Rrightarrow Q'_h(x,z) \qquad \forall x \in X, z.\ \mathcal{A} \models Q_a(f(x),z) \Rrightarrow Q'_a(x,z) \\ \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}y \in Y.\langle\ P_h \mid P_a(y)\ \rangle\ \mathbb{C}\ \exists z.\langle\ Q_h(y,z) \mid Q_a(y,z)\ \rangle\end{array}}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P'_a(x)\ \rangle\ \mathbb{C}\ \exists z.\langle\ Q'_h(x,z) \mid Q'_a(x,z)\ \rangle}$$

Figure 5.4: Proof rules: Logical rules

(Open region)

$$\frac{\forall x \in X.\ (x,z) \in \{(x,z) | x \in X \wedge R(x,z) \wedge (x,z) \in \mathcal{T}_{\mathbf{t}}(G(x))^*\} \qquad r \in dom(\mathcal{A}) \implies R = id \qquad \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P_a(x) \star I(\mathbf{t}_r^{\lambda}(x)) \star \lceil G(x) \rceil_r\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid \exists z.\ Q_a(x,y,z) \star I(\mathbf{t}_r^{\lambda}(z)) \star R(x,z)\ \rangle}{\Phi, \lambda+1, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid P_a(x) \star \mathbf{t}_r^{\lambda}(x) \star \lceil G(x) \rceil_r\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid \exists z.\ Q_a(x,y,z) \star \mathbf{t}_r^{\lambda}(z) \star R(x,z)\ \rangle}$$

(Update Region)

$$\frac{r \in dom(\mathcal{A}) \qquad \mathcal{A}' = \mathcal{A}[r \mapsto \bot] \qquad \mathcal{A}(r) = (\_, T) \qquad \Phi, \lambda, \mathcal{A}' \vdash \mathbb{V}x \in X \left\langle P_h \mid P_a(x) \star I(\mathbf{t}_r^{\lambda'}(x)) \right\rangle \mathbb{C}\ \exists y \left\langle \begin{array}{l}(\exists z \in T(x). \\ Q_h(x,y) \mid Q_a(x,y) \star \mathcal{I}(\mathbf{t}_r^{\lambda'}(z))) \\ \qquad \vee (Q'_a(x,y) \star \mathcal{I}(\mathbf{t}_r^{\lambda'}(x)))\end{array} \right\rangle}{\Phi, \lambda+1, \mathcal{A} \vdash \mathbb{V}x \in X. \left\langle P_h \mid P_a(x) \star \mathbf{t}_r^{\lambda'}(x) \star r \Mapsto \blacklozenge \right\rangle \mathbb{C}\ \exists y. \left\langle \begin{array}{l}(\exists z \in T(x). \\ Q_h(x,y) \mid\ Q_a(x,y) \star \mathbf{t}_r^{\lambda'}(z) \star r \Mapsto (x,z)) \\ \qquad \vee (Q'_a(x,y) \star \mathbf{t}_r^{\lambda'}(x) \star r \Mapsto \blacklozenge)\end{array} \right\rangle}$$

(Make atomic)

$$\frac{r \notin dom(\mathcal{A}) \qquad \forall x \in X.\ \mathcal{A} \models \mathbf{t}_r^{\lambda'}(x) \star \lceil G \rceil_r \text{ stable} \qquad \mathcal{A}' = \mathcal{A}[r \mapsto (X,T)] \qquad T \subseteq \mathcal{T}_t(G) \qquad \lambda' < \lambda \qquad \Phi, \lambda, \mathcal{A}' \vdash\ \{\ P_h \star \exists x \in X.\ \mathbf{t}_r^{\lambda'}(x) \star r \Mapsto \blacklozenge\ \}\ \mathbb{C}\ \{\exists x, y.\ T(x,y) \star Q_h(x,y)\ \star r \Mapsto (x,y)\ \}}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X.\langle\ P_h \mid \mathbf{t}_r^{\lambda'}(x) \star \lceil G \rceil_r\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid \mathbf{t}_r^{\lambda'}(y) \star \lceil G \rceil_r \star T(x,y)\ \rangle}$$

Figure 5.5: Proof rules: Atomicity rules

Chapter 6

Soundness

We prove the soundness of TaDA 2.0 against the semantic model we have described in Chapters 4 and 5. As the semantic model of TaDA is so far removed from that of TaDA 2.0, there is little to be gained in comparing their soundness proofs. However, the semantic model of TaDA 2.0 is closely related to that of TaDALive, so I complete all of the cases done explicitly in the TaDALive paper, and draw attention to the significance of the simplifications of TaDA 2.0 for the soundness proof.
I have proven the soundness of TaDA 2.0, Theorem 6.1, for more than half of the proof rules, including a number of each 'type' of rule, all of the proof rules for which a proof exists in TaDALive, and all of the proof rules to which I have made meaningful changes. The structure of this soundness proof is similar to that of TaDALive, but in the interest of conciseness, TaDALive omits all of the technical work required to fit together the structures of different components, which is nontrivia. For TaDA 2.0, this is provided explicitly in Appendix C and is intended to aid in developing the readers intuition for how the interactions of each component of the semantic model are tightly controlled in order to respect our intuition about the restrictions ghost states place on program executions.

Theorem 6.1 (TaDA 2.0 is sound)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$.

$$\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \textbf{ then } \models_\Phi \mathbb{C} : \mathbb{S}.$$

The proof proceeds by structural induction on the syntactic judgement $\vdash_\Phi \mathbb{C} : \mathbb{S}$, and in each case, requires proving that $\forall \varphi \in \mathsf{FImpl}.\ \models \varphi : \Phi \implies [\![\mathbb{C}]\!]_\varphi \subseteq [\![\mathbb{S}]\!]$. This requires detailed understanding of the types of traces in $[\![\mathbb{C}]\!]_\varphi$, particularly for the proof rules corresponding to primitive commands and Hoare commands which requires induction over the trace as well. I provide the soundness proof here of some illustrative cases, which I use to demonstrate how much of the surrounding technical work is used to reduce each case to a tractable statement (with the proofs of these lemmas in Appendix C). To aid the reader, we split Theorem 6.1 into a number of smaller theorems for each case.

6.1 Read

Theorem 6.2 (Soundness of READ)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\begin{array}{l}\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of READ} \\ \textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}.\end{array}$$

Proof. Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$, and that this holds by application of READ. We need to prove that $\models_\Phi \mathbb{C} : \mathbb{S}$, which by Definition 5.1.8 is that statement $\forall \varphi \in \mathsf{FImpl}, \models \varphi : \Phi \implies [\![\mathbb{C}]\!]_\varphi \subseteq [\![\mathbb{S}]\!]$. Take $\varphi \in \mathsf{FImpl}$ arbitrary and assume $\models \varphi : \Phi$. The command, $\mathbb{C}$, and specification, $\mathbb{S}$, from examination of READ (Section 5.2) must be

$$\mathbb{C} = \mathtt{x} := [\mathtt{E}]$$
$$\mathbb{S} = \Phi, \lambda, \mathcal{A} \vdash \forall n \in \mathbb{Z}. \left\langle \begin{array}{l}\mathtt{x} = x \star \\ \vec{\mathtt{w}} = \vec{w}\end{array} \;\middle|\; \mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \right\rangle \cdot \exists y. \left\langle \begin{array}{l}\mathtt{x} = n \star \\ \vec{\mathtt{w}} = \vec{w}\end{array} \;\middle|\; \mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \right\rangle_{\lambda, \mathcal{A}}$$

Our command and specification semantics (Definitions 4.1.3, 5.1.5) are traces - a finite sequence of program configurations. So we take $\tau \in [\![\mathbb{C}]\!]_\varphi$ and proceed by induction on traces, with different reasoning depending on if the step was made by the environment or the local thread. In each case, we aim to prove

that $\tau \in [\![\mathbb{S}]\!]$, i.e. that $\forall \sigma_l \in \mathsf{LStore}, v \in X$, if $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$ then $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$, where h_0 is the initial heap of τ, σ_0 is the initial PStore of τ, and

$$p_h = \mathcal{W}[\![\mathtt{x} = x \star \vec{\mathtt{w}} = \vec{w}]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$$
$$p_a = \lambda n.\ \mathcal{W}[\![\mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \star n \in \mathbb{Z}]\!]_{\mathcal{A}}^{\sigma_l}$$

I use these notational conventions freely in subsequent proofs.

If trace is a single state ($\tau = (\sigma_0, h_0, \mathbb{C})$), then take $\sigma_l \in \mathsf{LStore}, v \in \mathbb{Z}$ arbitrarily and assume

$$h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$$

The trace safety (Definition 5.1) rule TERM says $(\sigma_0, h_0, \mathbb{C}) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(p_h, p_a, v)\}$. That concludes this case.

Now we consider τ with exactly one step. The structure of the trace safety judgement is carefully designed to designate blame to either the local thread or environment thread, and is such that if the precondition of the specification is satisfied and the environment step is safe, then it will maintain the precondition. This means that it suffices only to consider traces which begin with a local step, and can freely prepend environment steps to cover the rest of the traces. This reasoning is captured in Lemma 6.1.

Lemma 6.1 (Suffices to consider traces beginning with a local step)**.** Let $\mathbb{C} \in \mathsf{Cmd}, \mathbb{S} \in \mathsf{Spec}$ and $\varphi \in \mathsf{FImpl}$ such that $\models \varphi : \Phi$. The following holds:

$$\begin{aligned} \textbf{if}\ \ & (\forall\ ((\sigma_0, h_0, \mathbb{C})\ s_0\ \tau) \in [\![\mathbb{C}]\!]_\varphi.\ \textbf{if}\ s_0 = \mathrm{loc}\ \textbf{then}\ \tau \in [\![\mathbb{S}]\!]) \\ & \textbf{then}\ (\forall \tau \in [\![\mathbb{C}]\!]_\varphi.\ \tau \in [\![\mathbb{S}]\!]) \end{aligned}$$

With this out of the way, let us consider τ of exactly one step, which is made by the local thread. Take $\sigma_l \in \mathsf{LStore}, v \in \mathbb{Z}$ arbitrarily and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$. The assumption on h_0 implies from the world satisfaction relation (Figure 4.1) and definition of p_a that $[\![\mathtt{E}]\!]_{\sigma_0} \in dom(h_0)$. Conclude that the only applicable local step in the operational semantics is the success case, (Figure B.1), and so $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark)$, with $\sigma_1 = \sigma_0[\mathtt{x} \mapsto h_0([\![\mathtt{E}]\!]_{\sigma_0})]$ and $h_1 = h_0$.

We aim to check that $\exists S \subseteq \mathcal{P}(\mathrm{SState}).\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$, by an application of LINPT (Figure 5.1). Let

$$q_h = \mathcal{W}[\![\mathtt{x} = v \star \vec{\mathtt{w}} = \vec{w}]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}.$$

To verify the premises of LINPT we need to apply the TERM rule to obtain

$$(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l, (q_h, p_a, \langle v, v \rangle), \{(q_h, p_a, \langle v, v \rangle)\}$$

Now we need to check

$$(h_0, h_1) \models_{\mathbb{S}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![\mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto v]\!]_{\mathcal{A}}^{\sigma_l} \tag{6.1}$$

Take $f \in View_{\mathcal{A}}$ arbitrary (as per the definition of frame-preserving updates, Definition 4.2.14) and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$. The lift of world composition to sets (Equation 4.1) and implied nonemptiness of p_h tells us that $\sigma_0(\mathtt{x}) = \sigma_l(x)$ and $\sigma_0(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$. From the definition of σ_1 it must be that $\sigma_1(\vec{\mathtt{w}}) = \vec{w}$. To find the value of $\sigma_1(\mathtt{x})$, find some $w_a \in p_a(v)$, $w_f \in f$ such that $h_0 \in \lfloor w_a \bullet w_f \rfloor_\lambda$ (world reification, Definition 4.2.13). From the definition of $p_a(v)$, find that $w_a \in \mathcal{W}[\![\mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto v]\!]_{\mathcal{A}}^{\sigma_l}$, and therefore $h([\![\mathtt{E}]\!]_{\sigma_0}) = v$. Therefore, $\sigma_1(\mathtt{x}) = v$, and so $q_h = \mathtt{Emp}_{\mathcal{A}}$. Conclude as $h_0 = h_1$ that $h_0 \in \lfloor w_a \bullet w_f \rfloor_\lambda$ implies $h_1 \in [\![q_h * \mathcal{W}[\![\mathtt{E}[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto v]\!]_{\mathcal{A}}^{\sigma_l} * f]\!]_\lambda$.

The final premises to check of LINPT is that $\checkmark = \checkmark \implies q_h = \mathcal{W}[\![\mathtt{x} = v \star \vec{\mathtt{w}} = \vec{w}]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$. This clearly true by construction. Therefore, application of LinPt yields $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(q_h, p_a, \langle v, v \rangle)\}$. Therefore $\tau \in [\![\mathbb{S}]\!]$.

Continue with the induction over τ - we need to check any subsequent steps maintain membership of $[\![\mathbb{S}]\!]$. From the operational semantics (Appendix B), $\checkmark$ is a stuck configuration for the local thread, that is, the local thread is considered to have terminated. Therefore, all subsequent steps of τ are environment steps. Consider $\tau = \tau' \mathrm{env} C$. As we discussed before, the specification semantics are carefully designed to be such that environment steps cannot themselves result in a program not satisfying a specification, i.e. we should be able to freely append any environment steps to a trace. This is captured in Lemma 6.2.

Lemma 6.2 (Trace safety is closed under appending env steps)**.** Let $\mathbb{S} \in \mathsf{Spec}$, τ env $C \in \mathsf{Trace}$, $\sigma_l \in \mathsf{LStore}$ and $(p_h, p_a, v) \in \mathsf{SState}$. The following holds:

$$\begin{aligned} &\textbf{if } \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S \\ &\quad \textbf{then } \exists S'.\ \tau \text{ env } C \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S' \end{aligned}$$

We use this Lemma in conjunction with the inductive hypothesis of $\tau' \in [\![\mathbb{S}]\!]$ to find that $\tau \in [\![\mathbb{S}]\!]$. Conclude that $\forall \tau \in [\![\mathbb{C}]\!]_\varphi$, $\tau \in [\![\mathbb{S}]\!]$, and so READ is sound.

□

Remark 6.1.1. In other cases of the soundness proof where we are required to check several steps of the local thread determined by the operational semantics, Lemma 6.2 suffices to handle all the interleaving environment steps as well.

Remark 6.1.2. The proof of Theorem 6.2 demonstrates that we cannot soundly introduce logical variables in the postcondition of specifications which are not bound in the precondition. Loosely, this is because we infer the values of logical variables from the $\sigma_l \in \mathsf{LStore}$ used to check the precondition, and as logical variables cannot change, we use the same σ_l to check the postcondition. This motivates the need for the additional σ_l component in the trace safety judgement, as well as the change in transformation of TaDA 2.0 between atomic triples and the hybrid triples they represent. Finally, READ should also demonstrate the difficulty which TaDALive was trying to express with managing atomically updated program variables: we need a logical variable to refer to their value in the atomic post-condition, but cannot simply create a new one. In the case of READ and FAS, the iniitally pseudoquantified logical variable suffices, but this is not the case for CAS, where the value of the atomically modified program variable is more nondeterministic. Here, it is absolutely essential to have the $\exists y$ in hybrid specifications to bound atomically modified logical variables to a value, despite the $\exists y$ not having a more unusual meaning (as implied in the original TaDA with the pseudo existential).

Remark 6.1.3. The properties of the trace safety captured by the above lemmas are relied upon in TaDALive for the soundness proof to hold, although never explicitly stated or proven. Also, their infinite traces would require the argument above to be coinductive.

6.2 Sequence

To prove soundness in the case the last rule applied is SEQ, we again need to understand the structure of the traces which can be members of $[\![\mathbb{C}_1\ ;\ \mathbb{C}_2]\!]_\varphi$. This reasoning is applied in the soundness proofs of other Hoare rules, such as WHILE, so the following lemmas are designed to be general enough to be applicable for other cases.

Definition 6.2.1 (Trace sequencing)**.** For $\mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}$ and $\tau \in \mathsf{Trace}_\varphi$, define $\tau_{;\mathbb{C}_2}$ to be the result of the following transformation:

$$\tau_{;\mathbb{C}_2} = \begin{cases} (\sigma, h, \mathbb{C}_1\ ;\ \mathbb{C}_2) & \tau = (\sigma, h, \mathbb{C}_1) \\ \tau'_{;\mathbb{C}_2}\ s_i\ (\sigma, h, \mathbb{C}_1\ ;\ \mathbb{C}_2) & \tau = \tau' s_i (\sigma, h, \mathbb{C}_1) \end{cases}$$

This is lifted to sets in the obvious way. For $\tau_1, \tau_2 \in \mathsf{Trace}_\varphi$,

$$\tau_1; \tau_2 = \begin{cases} \tau_{1;\mathbb{C}}\ s_i\ \tau_2 & \tau_2[0] = (\_, \_, \mathbb{C}) \wedge \tau_{1;\mathbb{C}}[-1] \xrightarrow{s_i}_\varphi \tau_2[0] \\ \bot & \text{otherwise} \end{cases}$$

where $\tau[-1]$ refers to the final program configuration of the trace. We lift this to sets of PTraces in the following manner:
For $\mathbb{T}_1, \mathbb{T}_2 \in \mathcal{P}(\mathsf{Trace}_\varphi)$,

$$\mathbb{T}_1; \mathbb{T}_2 = \begin{cases} \mathbb{T}_{1;\mathbb{C}} \cup \{\ \tau_1; \tau_2 \mid \tau_1 \in \mathbb{T}_1, \tau_2 \in \mathbb{T}_2, \tau_1; \tau_2 \neq \bot\ \} & \forall \tau \in \mathbb{T}_2, \tau[0] = (\_, \_, \mathbb{C}) \\ \emptyset & \text{otherwise} \end{cases}$$

This is only well-defined when every trace in $\mathbb{T}_2$ begins in a configuration with the same command.

Lemma 6.3 (Semantics of sequenced commands)**.** For $\mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}, \varphi \in \mathsf{FImpl}$,

$$[\![\mathbb{C}_1 \ ; \ \mathbb{C}_2]\!]_\varphi = [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$$

In order to use the above to prove soundness, we need some work to be able to use the premises of proof rules by transform a trace safety judgement of each part of the trace to one which verifies the safety of the combined trace. As part of this, to apply the definition of $[\![\mathbb{S}]\!]$ to find the trace safety judgement statement, we need to be able to argue about terminating traces ending in a configuration satisfying the postcondition.

Lemma 6.4 (Terminated traces satisfying precondition satisfy postcondition)**.** Let $\tau \in \mathsf{Trace}$, $\mathbb{S} \in \mathsf{Spec}$, $\sigma_l \in \mathsf{LStore}$, $p_h \in View_{\mathcal{A}}$ and $v \in X$. Then, for $p_a = \lambda x.\ \mathcal{W}[\![P_a(x)]\!]^{\sigma_l}_{\mathcal{A}}$ and $q_h = \mathcal{W}[\![Q_h]\!]^{\sigma_{i+1} \circ \sigma_l}_{\mathcal{A}}$, following holds:

$$\begin{aligned}&\textbf{if} \ \ h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda \ \textbf{and} \ \exists S.\ (S \text{ is nonempty } \textbf{and } \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S) \ \textbf{and} \\ &\quad \tau = \tau'(h_i, \sigma_i, \mathbb{C}_i) s_i (h_{i+1}, \sigma_{i+1}, \checkmark) \ \textbf{then} \ h_{i+1} \in [\![q_h * \mathrm{True}_{\mathcal{A}}]\!]_\lambda.\end{aligned}$$

Lemma 6.5 (Safety of concatenated traces)**.** For $\mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}, P, Q, R \in \mathsf{Assert}, \varphi \in \mathsf{FImpl}$ such that $\models \varphi : \Phi, \tau \in [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$, **if** $\models_\Phi \mathbb{C}_1 : \mathbb{S}_1$ **and** $\models_\Phi \mathbb{C}_2 : \mathbb{S}_2$, where

$$\begin{aligned}\mathbb{S}_1 &= \mathbb{V} x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle \ \cdot\ \exists y.\ \langle\ R \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}} \\ \mathbb{S}_2 &= \mathbb{V} x \in \mathsf{AVal}.\ \langle\ R \mid \mathsf{emp}\ \rangle \ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}\end{aligned}$$

then $\tau \in [\![\mathbb{S}]\!]$, where

$$\mathbb{S} = \mathbb{V} x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle \ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

Theorem 6.3 (Soundness of SEQ)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\begin{aligned}&\textbf{if} \ \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of SEQ} \\ &\quad \textbf{then} \ \models_\Phi \mathbb{C} : \mathbb{S}.\end{aligned}$$

Proof.

$$\begin{aligned}\mathbb{C} &= \mathbb{C}_1 \ ; \ \mathbb{C}_2 \\ \mathbb{S} &= \mathbb{V} x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle \ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}}\end{aligned}$$

Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$. Then we find some assertion R such that for

$$\begin{aligned}\mathbb{S}_1 &= \mathbb{V} x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle \ \cdot\ \exists y.\ \langle\ R \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}} \\ \mathbb{S}_2 &= \mathbb{V} x \in \mathsf{AVal}.\ \langle\ R \mid \mathsf{emp}\ \rangle \ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}}\end{aligned}$$

we have $\vdash_\Phi \mathbb{C}_1 : \mathbb{S}_1$ and $\vdash_\Phi \mathbb{C}_2 : \mathbb{S}_2$. By the inductive hypothesis we have

$$\models_\Phi \mathbb{C}_1 : \mathbb{S}_1 \tag{6.2}$$

$$\models_\Phi \mathbb{C}_2 : \mathbb{S}_2 \tag{6.3}$$

We need to prove $\models_\Phi \mathbb{C} : \mathbb{S}$, so take φ arbitrary and assume that $\models \varphi : \Phi$. Take $\tau \in [\![\mathbb{C}]\!]_\varphi$ arbitrary. From Lemma 6.3, $\tau \in [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$, and Lemma 6.5 gives the result. □

See how almost all of the heavy lifting of this proof is in Lemmas 6.3 and 6.5. As induction on the operational semantics of traces of other Hoare commands can reduce them to a statement about sequenced traces, these lemmas are directly application to those cases (such as WHILE).

Remark 6.2.1. TaDA 2.0's traces are sequences of program configurations, defined in Definition 4.1.2 as

$$c \in \mathrm{Conf} \triangleq (\mathsf{PStore} \times \mathsf{Heap} \times (\mathsf{Cmd} \uplus \checkmark)) \uplus \{\lightning\}$$

The equivalent type of trace in TaDALive is a program trace, of which the final component is not $\mathsf{Cmd} \uplus \checkmark$, but an inductive construction PState which approximately represents commands and locally defined program variables in nested program stores. This greatly complicates the inductive step of the soundness of Hoare rules, as subparts of traces which are produced by the operational semantics are still not in the semantics of TaDALive's commands (as this requires the initial state to represent a genuine command

and not a nested structure). This can be seen most explicitly by considering $\mathbb{C} = \texttt{var x = E in } \mathbb{C}'$, whose operational semantics in TaDALive produces a trace $(\sigma_0, h, \mathbb{C})\text{loc}(\sigma_0, h, ([\texttt{x} \mapsto [\![\texttt{E}]\!]_{\sigma_0}], \mathbb{C}'))\tau'$. It is immediately clear that $(\sigma_0, h, ([\texttt{x} \mapsto [\![\texttt{E}]\!]_{\sigma_0}], \mathbb{C}'))\tau'$ is not a member of $[\![\mathbb{C}']\!]_\varphi$, and thus the premise of the VAR rule is inapplicable to check safety of the rest of the trace. Furthermore, it would require a lot of technical work to get around this, likely in the form of some kind of very general trace transformation from $(\sigma_0, h, ([\texttt{x} \mapsto [\![\texttt{E}]\!]_{\sigma_0}], \mathbb{C}'))\tau'$ to $(\sigma_0[\texttt{x} \mapsto [\![\texttt{E}]\!]_{\sigma_0}], h, \mathbb{C}')\tau''$. TaDA 2.0 sidesteps this by providing a different operational semantics rule for $\texttt{var x = E in } \mathbb{C}'$, and as a result the premises of Hoare proof rules are immediately applicable. This materialises not just in the proof rule for VAR, but also SEQ, WHILE and PAR.

Remark 6.2.2. In TaDALive, all of the traces are infinite, and so the trace safety judgement is coinductive, rather than inductive. As a result, any of these lemmas with proofs via induction over the trace safety judgement, such as Lemma 6.5 (but is a common proof pattern occuring in most of the soundness proofs for logical proof rules) are by coinduction.

6.3 Parallel

Similar to in the SEQ case, we aim to determine the traces which are in $[\![\mathbb{C}_1 \;||\; \mathbb{C}_2]\!]_\varphi$ by defining how these traces are constructed from those of $[\![\mathbb{C}_1]\!]_\varphi$ and $[\![\mathbb{C}_2]\!]_\varphi$, using the $\bowtie$ operator. Definition 6.3.1 will be sufficient for this, although it seems restrictive in disallowing updates to the program store. Actually, the well-formedness requirement of the parallel command, $mod(\mathbb{C}_1 \;||\; \mathbb{C}_2) = \emptyset$, and Lemma 6.6 shows us that $\forall \tau \in [\![\mathbb{C}_1 \;||\; \mathbb{C}_2]\!], i \neq j.\ \sigma_i = \sigma_j$, so this is not a concern.

Lemma 6.6. For $\varphi \in \mathsf{FImpl}, \mathbb{C} \in \mathsf{Cmd}, \tau \in \mathsf{Trace}$,

$$\tau(\sigma_0, h_0, \mathbb{C}_0)s_0(\sigma_1, h_1, \mathbb{C}_1) \in [\![\mathbb{C}]\!]_\varphi \implies \forall \texttt{x} \in \mathsf{PVar} \setminus mod(\mathbb{C}).\ \sigma_0(\texttt{x}) = \sigma_1(\texttt{x})$$

Definition 6.3.1 (Bowtie)**.**

$$\begin{aligned}
(\sigma, h, \mathbb{C}_1) \bowtie (\sigma, h, \mathbb{C}_2) &= (\sigma, h, \mathbb{C}_1 \;||\; \mathbb{C}_2) \\
\tau_1'\text{loc}(\sigma, h, \mathbb{C}_1') \bowtie \tau_2'\text{env}(\sigma, h, \mathbb{C}_1) &= \\
&\quad (\tau_1' \bowtie \tau_2')\text{loc}(\sigma, h, \mathbb{C}_1' \;||\; \mathbb{C}_2) \\
\tau_1'\text{env}(\sigma, h, \mathbb{C}_1) \bowtie \tau_2'\text{loc}(\sigma, h, \mathbb{C}_2') &= \\
&\quad (\tau_1' \bowtie \tau_2')\text{loc}(\sigma, h, \mathbb{C}_1 \;||\; \mathbb{C}_2') \\
\tau_1'\text{env}(\sigma, h, \mathbb{C}_1) \bowtie \tau_2'\text{env}(\sigma, h, \mathbb{C}_2) &= \\
&\quad (\tau_1' \bowtie \tau_2')\text{env}(\sigma, h, \mathbb{C}_1 \;||\; \mathbb{C}_2)
\end{aligned}$$

where $\bowtie$ is undefined in all other cases. Lift this to sets as following:

$$\begin{aligned}
\mathbb{T}_1 \bowtie \mathbb{T}_2 \triangleq &\left\{ \tau_1 \bowtie \tau_2 \;\middle|\; \tau_1 \in \mathbb{T}, \tau_2 \in \mathbb{T} \right\} \cup \\
&\left\{ (\tau_1 \bowtie \tau_2)\text{loc}(\sigma, h, \checkmark) \;\middle|\; \begin{array}{l} \forall i \in \{1, 2\}.\ \tau_i \in \mathbb{T}_i \wedge \tau_i[-1] = (_, _, \checkmark) \wedge \\ (\tau_1 \bowtie \tau_2)\text{loc}(\sigma, h, \checkmark) \in \mathsf{Trace} \end{array} \right\}
\end{aligned}$$

Lemma 6.7 (Semantics of parallel commands is bowtie of semantics of each command)**.**

$$\forall \mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}, \varphi \in \mathsf{FImpl}.\ [\![\mathbb{C}_1 \;||\; \mathbb{C}_2]\!]_\varphi \subseteq [\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_2]\!]_\varphi$$

Again, similar to the SEQ case, we provide the following lemma to transform trace safety judgements of traces of the constituent parts of the parallel command into one from their combined perspective. This is where the bulk of the work for the soundness of PAR comes in. As usual, the proof is in Appendix C.

Lemma 6.8 (Safety of parallel traces)**.** For $\tau_1, \tau_2 \in \mathsf{Trace}, v \in \mathsf{AVal}', \sigma_l \in \mathsf{LStore}, S_1, S_2 \in \mathcal{P}(\text{SState})$, $p1_h, p2_h \in View_{\mathcal{A}}$ such that $\tau_1 \bowtie \tau_2$ is well-defined and $(\tau_1 \bowtie \tau_2)[0] = (h, \_, \_)$,

$$\begin{array}{l}
\textbf{if } \left(\begin{array}{c} \tau_1 \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v), S_1 \textbf{ and } \tau_2 \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2 \textbf{ and} \\ h \in [\![p1_h * p2_h * p1_a(v) * p2_a(v) * \text{True}_{\mathcal{A}}]\!]_\lambda \end{array} \right) \\
\textbf{then } \tau_1 \bowtie \tau_2 \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S
\end{array}$$

where

$$S = \{\ (q1_h * q2_h, p1_a * p2_a, v') \mid (q1_h, p1_a, v') \in S_1 \wedge (q2_h, p2_a, v') \in S_2\ \}$$

and $p1_a = p2_a = \lambda x.\texttt{Emp}_{\mathcal{A}}$, and I also write $p1_a * p2_a$ to denote $\lambda x.\ p1_a(x) * p2_a(x)$.

Finally, observe from the lift of $\bowtie$ to sets of traces that we have an additional case for the terminating step of a trace in $[\![\mathbb{C}_1 \;||\; \mathbb{C}_2]\!]_\varphi$. Unfortunately, this requires reasoning about the trace safety by appending rather than prepending execution steps. To this end, we provide a lemma on appending local steps to a trace.

Lemma 6.9 (Trace safety is closed under appending safe local steps)**.** Let $\mathbb{S} \in \mathsf{Spec}$, $f : \mathsf{SState} \to \mathit{View}_\mathcal{A}$, $g : \mathsf{SState} \to \mathsf{AVal}$ and $\tau(\sigma_1, h_1, \mathbb{C}_1)$ loc $(\sigma_2, h_2, \mathbb{C}_2) \in \mathsf{Trace}$. The following hold:

1. **if** $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), S$ **and**
$\forall (p'_h, p_a, v') \in S.\ (h_1, h_2) \models_{\lambda,\mathcal{A}} p'_h * p_a(v') \twoheadrightarrow f(p'_h, p_a, v') * p_a(v')$ **and**
(**if** $\mathbb{C}_2 = \checkmark$ **then** $f(p'_h, p_a, v') = \mathcal{W}[\![Q_h(v')]\!]_\mathcal{A}^{\sigma_l \circ \sigma_2}$ **and** $v' \in \mathsf{AVal} \times \mathsf{AVal}$)
then $\tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), \{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S\ \}$

2. **if** $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), S$ **and**
$\forall (p'_h, p_a, v') \in S.\ (h_1, h_2) \models_{\lambda,\mathcal{A}} p'_h * p_a(v') \twoheadrightarrow f(p'_h, p_a, v') * \mathcal{W}[\![Q_a(v, g(p'_h, p_a, v'))]\!]_\mathcal{A}^{\sigma_l}$ **and**
(**if** $\mathbb{C}_2 = \checkmark$ **then** $f(p'_h, p_a, v') = \mathcal{W}[\![Q_h(v')]\!]_\mathcal{A}^{\sigma_l \circ \sigma_2}$)
then $\tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), \{\ (f(p'_h, p_a, v'), p_a, g(p'_h, p_a, v')) \mid (p'_h, p_a, v') \in S\ \}$

Remark 6.3.1. The S component of the trace safety judgement is only needed to reason about satisfying the postcondition when appending to a trace. It is possible that an alternative proof of soundness may not require this, in which case the set of SStates could be removed entirely.

Checking that terminated threads satisfy their postcondition is intertwined within the trace safety judgement. The following is the final lemma we need which allows for reasoning about the postcondition while inducting on trace safety.

Lemma 6.10 (Safe terminated traces satisfy postcondition)**.** Let $\tau(\sigma_n, h_n, \mathbb{C}_n) \in \mathsf{Trace}$, $\mathbb{C} \in \mathsf{Cmd}$, $\mathbb{S} \in \mathsf{Spec}$, φ, $\sigma_l \in \mathsf{LStore}$, $v \in X$ and $S \in \mathcal{P}(\mathsf{SState})$. The following holds:

if $\tau(\sigma_n, h_n, \mathbb{C}_n) \in [\![\mathbb{C}]\!]_\varphi$ **and** $\mathbb{C}_n = \checkmark$ **and** $\tau(\sigma_n, h_n, \mathbb{C}_n) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), S$ **and** S nonempty
then $\forall (p'_h, p_a, v') \in S.\ p'_h = \mathcal{W}[\![Q_h(v')]\!]_\mathcal{A}^{\sigma_n \circ \sigma_l}$ **and** $v' \in \mathsf{AVal} \times \mathsf{AVal}$

We are ready to prove the soundness of PAR.

Theorem 6.4 (Soundness of PAR)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

if $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by application of PAR
then $\models_\Phi \mathbb{C} : \mathbb{S}$.

Proof.

$$\mathbb{C} = \mathbb{C}_1 \;||\; \mathbb{C}_2$$
$$\mathbb{S} = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ P_1 \star P_2 \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ Q_1 \star Q_2 \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$. Then for

$$\mathbb{S}_1 = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ P_1 \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ Q_1 \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$
$$\mathbb{S}_2 = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ P_2 \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ Q_2 \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

we have by the inductive hypothesis that

$$\models_\Phi \mathbb{C}_1 : \mathbb{S}_1 \tag{6.4}$$

and

$$\models_\Phi \mathbb{C}_2 : \mathbb{S}_2 \tag{6.5}$$

By Lemma 6.7, it suffices to prove that $[\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_1]\!]_\varphi \subseteq [\![\mathbb{S}]\!]$. Take $\tau \in [\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_1]\!]_\varphi$, $\sigma_l \in \mathsf{LStore}, v \in \mathsf{AVal}$ arbitrary, let $p1_h = \mathcal{W}[\![P_1]\!]_\mathcal{A}^{\sigma_l \circ \sigma}$, $p2_h = \mathcal{W}[\![P_2]\!]_\mathcal{A}^{\sigma_l \circ \sigma}$, $p1_a = p2_a = \lambda x.\ \mathtt{Emp}_\mathcal{A}$ and assume

$$h_0 \in [\![p1_h * p2_h * p1_a(v) * p2_a(v) * \mathrm{True}_\mathcal{A}]\!]_\lambda \tag{6.6}$$

As $\tau \in [\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_1]\!]_\varphi$ we begin with the case that $\exists \tau_1, \tau_2$ such that $\tau_1 \in [\![\mathbb{C}_1]\!]_\varphi$, $\tau_2 \in [\![\mathbb{C}_2]\!]_\varphi$ and $\tau = \tau_1 \bowtie \tau_2$. Observe from the definition of $\bowtie$ that τ, τ_1, τ_2 agree on the heap and store in each configuration. It must be the case that $h_0 \in [\![p1_h * p1_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$ and $h_0 \in [\![p2_h * p2_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$, and thus as the inductive hypothesis says $\tau_1 \in [\![\mathbb{S}_1]\!]$ and $\tau_2 \in [\![\mathbb{S}_2]\!]$, we find $\exists S_1, S_2.\ \tau_1 \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v), S_1$ and $\tau_2 \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2$. Thus, Lemma 6.8 gives us that $\exists S.\ \tau_1 \bowtie \tau_2 \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S$ and therefore $\tau \in [\![\mathbb{S}]\!]$.

In the case that

$$\tau \in \left\{ (\tau_1 \bowtie \tau_2)\mathrm{loc}(\sigma, h, \checkmark) \;\middle|\; \begin{array}{l} \forall i \in \{1,2\}.\ \tau_i \in \mathbb{T}_i \wedge \tau_i[-1] = (\_, \_, \checkmark) \wedge \\ (\tau_1 \bowtie \tau_2)\mathrm{loc}(\sigma, h, \checkmark) \in \mathsf{Trace} \end{array} \right\}$$

it must be that τ has the form $\tau'(\sigma, h, \checkmark \;\;||\;\; \checkmark)\mathrm{loc}(\sigma, h, \checkmark)$ (as $\bowtie$ implies its final command is of the form $C_1 \;||\; C_2$, only one operational semantic rule is applicable) and see by the same argument as above that $\exists S.\ \tau'(\sigma, h, \checkmark \;\;||\;\; \checkmark) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. I claim that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ for the same S and aim to apply Lemma 6.9. For $f : \mathrm{SState} \to View_{\mathcal{A}},\ (p'_h, p_a, v') \mapsto p'_h$, it is clear that $\forall (p'_h, p_a, v') \in \mathrm{SState}$, $(h, h) \models_{\lambda,\mathcal{A}} p'_h * p_a(v') \mathrel{-\!\!*} f(p'_h, p_a, v') * p_a(v')$. It remains to prove that

$$\forall (p'_h, p_a, v') \in S.\ p'_h = \mathcal{W}[\![Q_1 * Q_2]\!]^{\sigma \circ \sigma_l}_{\mathcal{A}} \wedge v' \in \mathsf{AVal} \times \mathsf{AVal} \tag{6.7}$$

(as $S = \{(f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S\}$) and then we can conclude from Lemma 6.9 that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ and we are done. In order to prove Equation 6.7, we need to explicitly reason about the form of S, so let's remind ourselves that from the definition of $\tau'(\sigma, h, \checkmark \;\;||\;\; \checkmark) = \tau_1 \bowtie \tau_2$, therefore that τ_1 and τ_2 terminate, and therefore by Lemma 6.10 it must be that

$$\forall (p'_h, p_a, v') \in S_1.\ p'_h = \mathcal{W}[\![Q_1]\!]^{\sigma \circ \sigma_l}_{\mathcal{A}} \wedge v' \in \mathsf{AVal} \times \mathsf{AVal} \tag{6.8}$$

$$\forall (p'_h, p_a, v') \in S_2.\ p'_h = \mathcal{W}[\![Q_2]\!]^{\sigma \circ \sigma_l}_{\mathcal{A}} \wedge v' \in \mathsf{AVal} \times \mathsf{AVal} \tag{6.9}$$

From Lemma 6.8, we find that S satisfies Equation 6.7 so a final application of Stutter is legal here and provides $\tau \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S$. □

6.4 Frame

As in the soundness of SEQ or PAR, the soundness proof of FRAME (and almost every other logical proof rule) requires an inductive proof over the structure of the trace safety lemma to convert a safety statement with respect to $\mathbb{S}'$ to one of $\mathbb{S}$. I include this inductive proof here as it illustrates this in a more manageable form than that of SEQ or PAR, and because it provides insight into why we require the premise $\forall x \in X.\ \mathcal{A} \models R_a(x)$ stable, which seems unreasonably strong at first sight.

Theorem 6.5 (Soundness of FRAME)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\begin{array}{l} \textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of FRAME} \\ \quad \textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}. \end{array}$$

Proof. Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$. Let

$$\mathbb{S} = \mathbb{V}x \in X.\ \langle\ P_h \star R_h \mid P_a(x) \star R_a(x)\ \rangle\ \cdot\ \exists y.\ \langle\ Q_h(x,y) \star R_h \mid P_a(x,y) \star R_a(x)\ \rangle_{\lambda,\mathcal{A}}$$
$$\mathbb{S}' = \mathbb{V}x \in X.\ \langle\ P_h \mid P_a(x)\ \rangle\ \cdot\ \exists y.\ \langle\ Q_h(x,y) \mid P_a(x,y)\ \rangle_{\lambda,\mathcal{A}}$$

Then by assumption

$$mod(\mathbb{C}) \cap pv(R_h) = \emptyset \tag{6.10}$$
$$mod(\mathbb{C}) \cap pv(R_a(x)) = \emptyset \tag{6.11}$$
$$\mathcal{A} \models R_h \text{ stable} \tag{6.12}$$
$$\forall x \in X.\ \mathcal{A} \models R_a(x) \text{ stable} \tag{6.13}$$
$$\vdash_\Phi \mathbb{C} : \mathbb{S}' \tag{6.14}$$

From our inductive hypothesis, we have

$$\models_\Phi \mathbb{C} : \mathbb{S}' \tag{6.15}$$

Lemma 6.11. For $\tau \in [\![\mathbb{C}]\!]_\varphi, v \in X, \sigma_l \in \mathsf{LStore}$,

$$\begin{array}{l}\textbf{if } \exists S'.\ \tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), S' \textbf{ and } h_0 \in [\![p_h * r_h * p_a(v) * r_h(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda \\ \textbf{then } \exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h * r_h, p_a * r_a, v), S\end{array}$$

where

$$S \subseteq \{\ (q_h * r_h, p_a * r_a, v') \mid (q_h, p_a, v') \in S'\}$$

and $p_h = \mathcal{W}[\![P_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$, $p_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star x \in X]\!]_{\mathcal{A}}^{\sigma_l}$, $r_a = \lambda x.\ \mathcal{W}[\![R_a(x) \star x \in X]\!]_{\mathcal{A}}^{\sigma_l}$ and $r_h = \mathcal{W}[\![R_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$. Denote by $p_a * r_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star R_a(x) \star x \in X]\!]_{\mathcal{A}}^{\sigma_l}$.

Proof. Observe that $p_h * r_h = \mathcal{W}[\![P_h \star R_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$. Take $\tau \in [\![\mathbb{C}]\!]_\varphi, v \in X, \sigma_l \in \mathsf{LStore}$ arbitrary, find S' such that $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), S'$ and $h_0 \in [\![p_h * r_h * p_a * r_h * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$. Proceed by induction on the structure of $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), S'$

- **Case: Term.**
  If τ is safe by an application of Term, then it must be we have $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), \{\ (p_h, p_a, v)\}$ and that τ is a single state, thus an application of Term also gives $\tau \models_{\mathbb{S}} \sigma_l, (p_h * r_h, p_a * r_a, v), \{(p_h * r_h, p_a * r_a, v)\}$.

- **Case: Stutter.** I do the Stutter case - the LinPt case is identical.
  Assume $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), S'$ by an application of Stutter, so in fact

  $$\tau = (\sigma_0, h_0, \mathbb{C}_0)\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau'$$

  From the premises it must be that $(h_0, h_1) \models_{\lambda,\mathcal{A}} p'_h * p_a(v) \twoheadrightarrow p'_h * p_a(v)$ for some p'_h and that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p'_h, p_a, v), S'$. From the inductive hypothesis we find that

  $$(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}} \sigma_l, v, (p'_h * r_h, p_a * r_a, v), S$$

  where

  $$S \subseteq \{\ (q_h * r_h, p_a * r_a, v') \mid (q_h, p_a, v') \in S'\} \tag{6.16}$$

  From Lemma 4.5, and the stability conditions of r_h and r_a, find that

  $$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * r_h * p_a(v) * r_a(v) \twoheadrightarrow p'_h * r_h * p_a(v) * r_a(v)$$

  Again from the premises of Stutter, if $\mathbb{C}_1 = \checkmark$ then $p'_h = \mathcal{W}[\![Q_h(v)]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$. Given that

  $$p'_h * r_h = \mathcal{W}[\![Q_h(v')]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l} * \mathcal{W}[\![R_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$$

  and $\mathcal{W}[\![R_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l} = \mathcal{W}[\![R_h]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$ (from Equation 6.10), find

  $$p_h * r_h = \mathcal{W}[\![Q_h(v) \star R_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$$

  Finally, reapply Stutter to conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p_h * r_h, p_a * r_a, v), S$, with S given by Equation 6.16.

- **Case: Env.** I do the Env case, the Env' case is simpler.
  Assume $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), S$ by an application of Env, i.e.

  $$\tau = (\sigma, h_0, \mathbb{C})\mathrm{env}(\sigma, h_1, \mathbb{C})\tau'$$

  and from the premises of Env,

  $$\forall v' \in X.\ E(v') \implies (\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v'), S'_{v'}$$

  where

  $$E(v') \triangleq \exists p_e, p'_e.\ h_1 \in [\![p_h * p_a(v) * p_e]\!]_\lambda \wedge (h_1, h_2) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p'_e$$

  Let

  $$\begin{array}{rl} F(v') \triangleq \exists p_s, p'_s.\ & h_1 \in [\![p_h * r_h * p_a(v) * r_a(v) * p_s]\!]_\lambda \wedge \\ & (h_1, h_2) \models_{\lambda,\mathcal{A}} p_a(v) * r_a(v) * p_s \twoheadrightarrow p_a(v') * r_a(v') * p'_s \end{array}$$

I aim to prove

$$\forall v' \in X.\ F(v') \implies (\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}'} \sigma_l, (p_h * r_h, p_a * r_a, v'), S_{v'}$$

where

$$S_{v'} = \{\ (q_h * r_h, p_a * r_a, v') \mid (q_h, p_a, v') \in S_{v'}\ \}$$

Take $v' \in X$ arbitrary and assume $F(v')$. By taking $p_e = p_s * r_h * r_a(v)$, $p'_e = p'_s * r_h * r_a(v)$ we find $E(v')$, so from the premise of Env we have $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v'), S'_{v'}$ and applying the inductive hypothesis finds $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h * r_h, p_a * r_a, v'), S_{v'}$, with the necessary relation between $S_{v'}$ and $S'_{v'}$. The other premises of Env carry over to result in $(\sigma, h_1, \mathbb{C})\text{env}(\sigma, h_2, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h * r_h, p_a * r_a, v), S$ where S is the union of the earlier $S_{v'}$s.

- **Case: Env$\lightning$.**
  If $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), S'$ by an application of Env$\lightning$, then in fact we have

  $$(\sigma, h, \mathbb{C})\text{env}\lightning\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p_a, v), \emptyset$$

  By Env$\lightning$ we also find $(\sigma, h, \mathbb{C})\text{env}\lightning\tau' \models_{\mathbb{S}} \sigma_l, (p_h * r_h, p_a * r_a, v), \emptyset$.

□

Now we are ready to prove the result. Take $\tau \in [\![\mathbb{C}]\!], \sigma_l \in \mathsf{LStore}, v \in X$ arbitrary, and assume

$$h_0 \in [\![p_h * r_h * p_a * r_h * \text{True}_{\mathcal{A}}]\!]_\lambda$$

From our inductive hypothesis, $\tau \in [\![\mathbb{S}']\!]$, and from our assumption, $h_0 \in [\![p_h * p_a * \text{True}_{\mathcal{A}}]\!]_\lambda$. So

$$\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$$

Lemma 6.11 tells us $\exists S'.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S'$. Conclude $\tau \in [\![\mathbb{S}]\!]$.

□

Remark 6.4.1. In the STUTTER case of Lemma 6.11, we need to use Lemma 4.5 to conclude that the frame-preserving update of the premise of the trace safety judgement is transformable into a frame-preserving update with worlds representing the framed components R_h and $R_a(x)$. This is why it is crucial for the premise $\forall x \in X.\ \mathcal{A} \models R_a(x)$ stable to hold - without it, we wouldn't even be able to ensure that the abstract state of the region hasn't changed. I initially investigated whether an $\exists x \in X.\ R_a(x)$ stable requirement would suffice, but unfortunately we simply do not have the technical machinery at present to express a requirement of the x in $R_a(x)$ and $P_a(x)$ to agree, which is necessary to ensure the value of x in $R_a(x)$ is not modified in such an update. In practice, this greatly restricts any dependence on x of $R_a(x)$, but whether this restriction poses a genuine reduction in expressivity of TaDA 2.0 is unclear.

Chapter 7

Spinlock

We verify the following standard implementation of a spinlock.

$$\texttt{let } \mathit{lock}(\texttt{x}) = \\ \quad \texttt{var d} = 0 \texttt{ in} \\ \quad\quad \texttt{while } (\texttt{d} = 0) \\ \quad\quad\quad \texttt{d} := \texttt{CAS}(\texttt{x}, 0, 1)$$

$$\texttt{let } \mathit{unlock}(\texttt{x}) = \\ \quad [\texttt{x}] := 0$$

$$\texttt{let } \mathit{makelock}() = \\ \quad \texttt{var x} = 0 \texttt{ in} \\ \quad\quad \texttt{x} := \texttt{new}(1) ; \\ \quad\quad [\texttt{x}] := 0 ; \\ \quad\quad \texttt{ret} := \texttt{x}$$

Figure 7.1: Spinlock implementation

The heap location of x behaves as we would expect of a lock: it's never the case that a thread can't be sure if it has exclusive access to the lock, as the atomicity of the CAS provides absolute certainty that the lock was unlocked (heap location had value 0) at the time of locking (updating to heap value 1), and that no other thread could have interrupted and locked the lock in betweeen the thread checking this value and performing the update. Observe again here that this only requires the value retrieved from the heap to be atomic with the operation, and not the setting of the program variable to that value to be atomic with the heap operation (as discussed in Section 5.2). Therefore, our new primitive proof rules should suffice for atomicity behaviour such as this. Let us define the components of the ghost state needed to verify the behaviour of the lock.

The lock is intended to be used for threads to compete for shared resources, so it is crucial that it is shareable. Therefore, define a region $\mathbf{spin}_r(x, l)$, with syntactic region interpretation $\mathcal{I}(\mathbf{spin}^\lambda_r(x, l)) \triangleq x \mapsto l$. Define its semantic interpretation $\mathcal{I}_{\mathbf{spin}}[\![r, \lambda, (x, l)]\!]$ to be $\mathcal{W}[\![x \mapsto l]\!]^\emptyset_\mathcal{A}$ This clearly corrresponds to the concrete resources used in the implementation of the lock, and satisfies the stability requirement on region interpretations.
Its guard algebra is $(\{\mathbf{0}, \mathbf{E}\}, \bullet, \{\mathbf{0}\})$, with $\mathbf{0} \bullet \mathbf{0} = \mathbf{0}$, $\mathbf{0} \bullet \mathbf{E} = \mathbf{E} = \mathbf{E} \bullet \mathbf{0}$ and $\mathbf{E} \bullet \mathbf{E} = \bot$, i.e. we have one exclusive guard $\lceil E \rceil_r$, used to complete the logically atomic actions of locking and unlocking. The region interference function $\mathcal{T}_{\mathbf{spin}}(\mathbf{E})$ is the reflexive, transitive closure of $\{(0, 1), (1, 0)\}$ (and $\mathcal{T}_{\mathbf{spin}}(\mathbf{0}) = \emptyset$).

We aim to prove the following specifications:

$$\Phi, \lambda + 1, \mathcal{A} \vdash \{\texttt{emp}\}\ \text{makelock}()\ \{\exists r.\ \mathbf{spin}^\lambda_r(\texttt{ret}, 0) \star \lceil \mathbf{E} \rceil_r\}$$

$$\Phi, \lambda + 1, \mathcal{A} \vdash \forall l \in \{0, 1\}.\ \left\langle \mathbf{spin}^\lambda_r(\texttt{x}, l) \star \lceil \mathbf{E} \rceil_r \right\rangle\ \text{lock}(\texttt{x})\ \left\langle\ \mathbf{spin}^\lambda_r(\texttt{x}, 1) \star \lceil \mathbf{E} \rceil_r \star l = 0\ \right\rangle$$

$$\Phi, \lambda + 1, \mathcal{A} \vdash \left\langle\ \mathbf{spin}^\lambda_r(\texttt{x}, 1) \star \lceil \mathbf{E} \rceil_r \right\rangle\ \text{unlock}(\texttt{x})\ \left\langle\ \mathbf{spin}^\lambda_r(\texttt{x}, 0) \star \lceil \mathbf{E} \rceil_r\ \right\rangle$$

where $r \notin dom(\mathcal{A})$.

Briefly consider the precise semantics of this specification, to provide some intuition for why the exclusive guard does not prohibit sharing the lock with other threads. Section 9.2 provides guidance on how to use the lock for a client module.
After the call to makelock, the thread creating the lock has true ownership over the heap resource representing the shared location. This is represented by returning the shared region in the Hoare (or

private) part of the postcondition with the exclusive guard - there is no potential for race conditions on the resource until the creating thread has made it available to others. This represents genuinely exclusive ownership of the region.

Lock and unlock are both abstractly atomic actions with the shared region and the exclusive guard in the precondition. Here it is really essential to not be thinking of the atomic pre- and post-conditions as partial ownership - instead they represent resources that the thread knows to exist, and that will claim *exclusive* ownership over at the linearisation point. This intuition is key to understanding why the exclusive guard is does not prevent the sharing of the lock with other threads (clearly this would defeat the purpose of the lock as it would not be able to be used to compete for access to other shared resources). Finally, the lock specification mirrors that discussed in Section 3.1, with the pseudoquantifier being used to accept environment interference and restrict the value of the shared resource at the linearisation point, instead of in the precondition.

Makelock

We begin with the makelock deriviation, the proof sketch for which is given in Figure 7.2, demonstrating the standard ownership based reasoning in TaDA 2.0 and how to create shared regions out of locally owned resources. The proof of makelock is fairly standard in separation logic style proofs, as throughout the code the thread has true ownership over the resources. The only detail in the proof which provides insight into the workings of TaDA 2.0 is the use of the viewshift in the final application of cons, to update the logical view of the world from that of exclusively owned shared resources to that of a shared region with the same concrete representation:

$$\lambda+1, \mathcal{A} \models \mathtt{ret} = \mathtt{x} \star \mathtt{x} \mapsto 0 \Rrightarrow \exists r.\ \mathbf{spin}_r^\lambda(\mathtt{ret}, 0) \star \lceil E \rceil_r$$

Using Definitions 4.2.15 (Viewshift) and 4.2.14 (Frame-preserving update), if we take an arbitrary $\varsigma \in \mathsf{Store}, h \in \mathsf{Heap}, f \in View_{\mathcal{A}}$, and assume that $h \in [\![\mathcal{W}[\![\mathtt{ret} = \mathtt{x} \star \mathtt{x} \mapsto 0]\!]^\varsigma_{\mathcal{A}} \bullet f]\!]_{\lambda+1}$, so find some $w_h \in \mathcal{W}[\![\mathtt{ret} = \mathtt{x} \star \mathtt{x} \mapsto 0]\!]^\varsigma_{\mathcal{A}}, w_f \in f$ such that $h \in \lfloor w_h \bullet w_a \rfloor_{\lambda+1}$. From the world semantics, it must be that $\varsigma\mathtt{ret} = \varsigma\mathtt{x}$, and furthermore from the region interpretation of spinlock, that $w_h \in \mathcal{I}_{\mathbf{spin}}[\![r, \lambda, (\mathtt{ret}, 0)]\!]$. As $f \in \mathtt{World}^\uparrow_{\mathcal{A}}$, there must be some w'_f adding the region $\mathbf{spin}_r^\lambda(\mathtt{ret}, 0)$ to its ρ such that $w'_f \in f$, so we can find that $h \in [\![\mathcal{W}[\![\exists r.\ \mathbf{spin}_r^\lambda(\mathtt{ret}, 0)]\!]^\varsigma_{\mathcal{A}} * f]\!]_{\lambda+1}$. This illustrates how we can create shared regions out of exclusively owned resources. Finally, to conclude that h represents some world consistent with owning the guard $\lceil E \rceil_r$, observe that when we added $\mathbf{spin}_r^\lambda(\mathtt{ret}, 0)$ to w_h and w_f (finding some $w'_h \in \mathcal{W}[\![\mathbf{spin}_r^\lambda(\mathtt{ret}, 0)]\!]^\varsigma_{\mathcal{A}}$ and $w'_f \in f$), we are implicitly updating both γ's to represent the zero guard of the guard algebra $\mathcal{G}_{\mathbf{spin}}$. Therefore, w'_f is consistent with the local thread owning the exclusive guard, so we can find some $w_g \in \mathcal{W}[\![\lceil E \rceil_r]\!]^\varsigma_{\mathcal{A}}$ such that $h \in \lfloor w'_h \bullet w_g \bullet w_f \rfloor_{\lambda+1}$ and conclude that $h \in [\![\mathcal{W}[\![\exists r.\ \mathbf{spin}_r^\lambda(\mathtt{ret}, 0) \star \lceil E \rceil_r]\!]^\varsigma_{\mathcal{A}} * f]\!]_{\lambda+1}$. This gives the premise we needed to apply CONS to the exclusively owned resources to find the shared region and guard in the postcondition of makelock.

Unlock

Unlock will show us how we use an atomic operation to modify a shared region (Figure 7.3). It uses OPEN REGION to update the abstract state of the region corresponding to the concrete update in the code of the region interpretation.

(Open region)

$$\frac{\forall x \in X.\ (x,z) \in \{(x,z) | x \in X \wedge R(x,z) \wedge (x,z) \in \mathcal{T}_{\mathbf{t}}(G(x))^*\} \qquad r \in dom(\mathcal{A}) \implies R = id \qquad \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}x \in X. \langle\ P_h \mid P_a(x) \star I(\mathbf{t}_r^\lambda(x)) \star \lceil G(x) \rceil_r\ \rangle\ \mathbb{C}\ \exists y. \langle\ Q_h(x,y) \mid \exists z.\ Q_a(x,y,z) \star I(\mathbf{t}_r^\lambda(z)) \star R(x,z)\ \rangle}{\Phi, \lambda+1, \mathcal{A} \vdash \mathbb{V}x \in X. \langle\ P_h \mid P_a(x) \star \mathbf{t}_r^\lambda(x) \star \lceil G(x) \rceil_r\ \rangle\ \mathbb{C}\ \exists y. \langle\ Q_h(x,y) \mid \exists z.\ Q_a(x,y,z) \star \mathbf{t}_r^\lambda(z) \star R(x,z)\ \rangle}$$

The use of the rule in the unlock derivation is

$$\frac{\forall y \in \{1\}.\ (y,z) \in \{(y,z) | y \in \{1\} \wedge z = 0 \wedge (y,z) \in \mathcal{T}_{\mathbf{spin}}(E)^*\} \qquad r \in dom(\mathcal{A}) \implies R = id \qquad \Phi, \lambda, \mathcal{A} \vdash \mathbb{V}y \in \{1\}. \langle\ \mathtt{x} = x \mid x \mapsto y \star \lceil E \rceil_r\ \rangle\ \mathbb{C}\ \langle\ \mathtt{x} = x \mid \exists z.\ x \mapsto z \star z = 0\ \rangle}{\Phi, \lambda+1, \mathcal{A} \vdash \mathbb{V}y \in \{1\}. \langle\ \mathtt{x} = x \mid \mathbf{spin}_r^\lambda(x,y) \star \lceil E \rceil_r\ \rangle\ \mathbb{C}\ \langle\ \mathtt{x} = x \mid \exists z.\ \mathbf{spin}_r^\lambda(x,z) \star z = 0\ \rangle}$$

where by assumption $r \notin dom(\mathcal{A})$ and $(1,0) \in \mathcal{T}_{\mathbf{spin}}(E)$. This illustrates how to take a logically atomic step (which does not necessarily have to be primitive atomic) to update a shared region, when we hold the permissions for such an action. The atomicity of the step is crucial to preventing race-conditions, as there is no assumption on exclusive access to the resources. In the case of unlocking a spinlock, the action is actually primitive atomic, so we do not need to use any of the other atomicity rules to verify this - the existing mutate rule does the job.

$\Phi, \lambda+1, \mathcal{A} \vdash \mathbb{V} y \in \mathsf{AVal}.\ \langle\ \mathsf{emp}\ |\ \mathsf{emp}\ \rangle$

$\texttt{var x} = 0\ \texttt{in}$

// Var.

// $\texttt{x} \notin fv(P_h) \cup fv(Q_h) \cup fv(\texttt{E})$ where $P_h = Q_h = \mathsf{emp}$ and $\texttt{E} = 0$

$\mathbb{V} y \in \mathsf{AVal}.\ \langle\ \texttt{x} = 0\ |\ \mathsf{emp}\ \rangle$

Cons, ∃Elim:

- // $\lambda+1, \mathcal{A} \models \texttt{x} = 0 \Rrightarrow \exists x.\ 1 \geq 0 \star \texttt{x} = x$ and $\mathcal{A} \models \texttt{x} = 0$ stable
- $\langle\ 1 \geq 0 \star \texttt{x} = x\ |\ \mathsf{emp}\ \rangle$
- $\texttt{x} := \texttt{new}(1)$
- $\langle\ \exists y, v.\ \texttt{x} = y \star \circledast_{0 \leq i \leq 0}\, y + i \mapsto v\ |\ \mathsf{emp}\ \rangle$

// $\lambda+1, \mathcal{A} \models \exists y, v.\ \texttt{x} = y \star \circledast_{0 \leq i \leq 0}\, y + i \mapsto v \Rrightarrow \exists v.\ \texttt{x} \mapsto v$ and

// $\mathcal{A} \models \exists v.\ \texttt{x} = v$ stable

$\langle\ \exists v.\ \texttt{x} \mapsto v\ |\ \mathsf{emp}\ \rangle$

// Seq.

$\mathbb{V} y \in \mathsf{AVal}.\ \langle\ \exists v.\ \texttt{x} \mapsto v\ |\ \mathsf{emp}\ \rangle$

cons, ∃Elim:

- // $\lambda+1, \mathcal{A} \models \exists v.\ \texttt{x} \mapsto v \Rrightarrow \exists v, x.\texttt{x} = x \star x \mapsto v$ and
- // $\mathcal{A} \models \exists v.\ \texttt{x} = v$ stable
- $\langle\ \texttt{x} = x \star x \mapsto v |\ \mathsf{emp}\ \rangle$

  Atomic-weaken, Subst:

  - // $\mathcal{A} \models \texttt{x} = x \star x \mapsto v$ stable
  - // $f : \mathsf{AVal} \to \mathbb{N}$ is the constant function whose image is v.
  - // $\forall y \in \mathsf{AVal}, z.\ \lambda+1, \mathcal{A} \models x \mapsto v \iff x \mapsto f(y)$
  - $\mathbb{V} y \in \mathbb{N}.\ \langle\ \texttt{x} = x |\ x \mapsto v\ \rangle$
  - $[\texttt{x}] := 0$
  - // Mutate.
  - $\langle\ \texttt{x} = x |\ x \mapsto 0\ \rangle$
- // $\mathcal{A} \models x \mapsto 0$ stable
- $\langle\ \texttt{x} = x \star x \mapsto 0\ |\ \mathsf{emp}\ \rangle$

// $\forall y \in \mathsf{AVal}, z.\ \lambda+1, \mathcal{A} \models \texttt{x} = x \star x \mapsto 0 \Rrightarrow \texttt{x} \mapsto 0$ and

// $\mathcal{A} \models \texttt{x} \mapsto 0$ stable

$\langle\ \texttt{x} \mapsto 0\ |\ \mathsf{emp}\ \rangle$

// Seq.

$\mathbb{V} y \in \mathsf{AVal}.\ \langle\ \texttt{x} \mapsto 0\ |\ \mathsf{emp}\ \rangle$

Cons:

- $\langle\ \texttt{x} \mapsto 0\ |\ \mathsf{emp}\ \rangle$

  Frame:

  - // $\mathcal{A} \models \texttt{x} \mapsto 0$ stable and $mod(\texttt{ret} := \texttt{x}) \cap \{\texttt{x}\} = \emptyset$
  - $\langle\ \mathsf{emp}\ |\ \mathsf{emp}\ \rangle$
  - $\texttt{ret} := \texttt{x}$
  - // Assign.
  - $\langle\ \texttt{ret} = \texttt{x}\ |\ \mathsf{emp}\ \rangle$
- $\langle\ \texttt{ret} = \texttt{x} \star \texttt{x} \mapsto 0\ |\ \mathsf{emp}\ \rangle$

// $\lambda+1, \mathcal{A} \models \texttt{ret} = \texttt{x} \star \texttt{x} \mapsto 0 \Rrightarrow \exists r.\ \mathbf{spin}_r^{\lambda}(\texttt{ret}, 0) \star \lceil E \rceil_r$ and

// $\mathcal{A} \models \exists r.\ \mathbf{spin}_r^{\lambda}(\texttt{ret}, 0) \star \lceil E \rceil_r$ stable

$\langle\ \exists r.\ \mathbf{spin}_r^{\lambda}(\texttt{ret}, 0) \star \lceil E \rceil_r\ |\ \mathsf{emp}\ \rangle$

Figure 7.2: Make lock proof sketch for spin lock

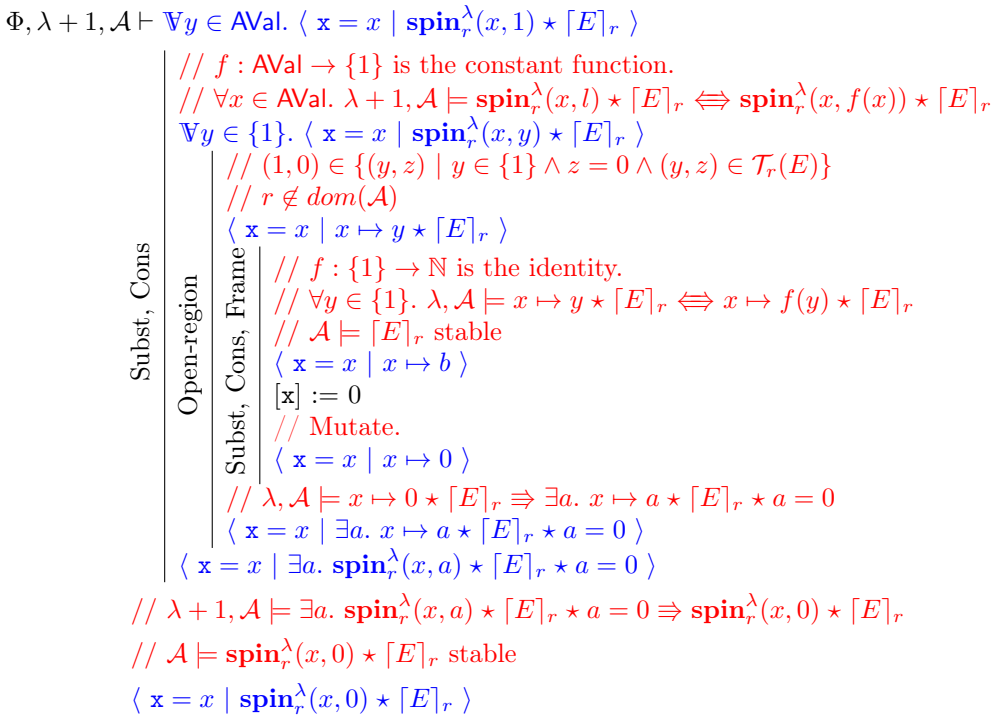

Figure 7.3: Unlock proof sketch for spin lock

Lock

Lock is the most interesting proof here, as it demonstrates how to prove that a logically atomic action indeed only represents one update of the abstract state. In order to prove that the function is atomic, we have to use the make atomic rule, and then dispatch the actual update to the shared region using update region. The full proof sketch is in Figure 7.4, with some additional premises made explicit in Figure 7.5.

In the derivation, make atomic is used to verify that `while` $(\mathtt{d} = 0)$ $\mathtt{d} := \mathtt{CAS}(\mathtt{x}, 0, 1)$ is an abstractly atomic operation, i.e. updates the abstract state l of $\mathbf{spin}_r^\lambda(\mathtt{x}, l)$ only once (represented by the heap resource $\mathtt{x} \mapsto l$). Make atomic takes the guard from the pre- and post-condition and replaces it with the atomicity tracking resource, representing permission to perform the same operation as the guard (in our case, $\{(0,1)\}$), but preventing the guard from being used to perform more than one update to the lock (for example, first unlocking the lock, and then relocking it would not be verifiable here). The stability premise of make atomic is what absolutely necessitates the guard associated with the lock to be exclusive. Otherwise, we could not be certain that the environment can't interfere with something in the midst of performing our logically atomic action. However, it is sufficient for this exclusivity to hold only at the linearisation point in make atomic, which is how we will be able to share the lock with other threads.

The only way to use the atomicity tracking resource to perform an update is via the update region rule. In our case, this is the mechanism for using the concrete update to heap location `x` to justify an update to the abstract state of the lock (l updating from 0 to 1). Furthermore, the disjunction in the update region postcondition is actually necessary for the components of this derivation to correctly fit together - otherwise, the nondeterministic nature of CAS would prevent us from using update region. Finally, it can be observed that the *new* CAS rule, with the atomicity separation of the store and heap update is sufficient here for the derivation.

One more interesting technical detail of the proof is the way we discard the shared region to match the premise of make atomic. Make atomic does not require any longevity of the shared region after the linearisation (in keeping with our intuition that after the linearisation point, we cannot make any assumptions on how the environment will act on it), but in our case we actually do maintain some notion of ownership of the shared region. Therefore, being able to use viewshift to discard the shared region from the assertions is crucial here.

Figure 7.4: Lock proof sketch for spin lock

We have the following additional side conditions for rules applied in the lock proof sketch.

$$(1) : \lambda+1, \mathcal{A}' \models \mathtt{x} = x \star \mathtt{d} = 0 \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(x,1) \star r \Mapsto \blacklozenge \Rrightarrow$$
$$\mathtt{x} = x \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\mathtt{x},l) \star \begin{pmatrix}(\mathtt{d}=0 \wedge r \Mapsto \blacklozenge) \vee \\ (\mathtt{d}=1 \wedge r \Mapsto (0,1))\end{pmatrix} \text{ and}$$
$$\mathcal{A}' \models \mathtt{x} = x \star \mathtt{d} = 0 \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(x,1) \star r \Mapsto \blacklozenge \text{ stable}$$

$$(2) : \lambda+1, \mathcal{A}' \models \mathtt{x} = x \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\mathtt{x},l) \star \begin{matrix}(\mathtt{d}=0 \wedge r \Mapsto \blacklozenge) \vee \\ (\mathtt{d}=1 \wedge r \Mapsto (0,1) \wedge l = 0)\end{matrix} \star \mathtt{d} = 0 \Rrightarrow$$
$$\mathtt{x} = x \star \mathtt{d} = 0 \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\mathtt{x},l) \star r \Mapsto \blacklozenge \text{ and}$$
$$\mathcal{A}' \models \mathtt{x} = x \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\mathtt{x},l) \star \begin{matrix}(\mathtt{d}=0 \wedge r \Mapsto \blacklozenge) \vee (\mathtt{d}=1 \wedge \\ r \Mapsto (0,1) \wedge l = 0)\end{matrix} \star \mathtt{d} = 0 \text{ stable}$$

$$(3) : \lambda, \mathcal{A} \models \begin{matrix}(l = 0 \wedge y = 1 \star x \mapsto 1) \vee \\ (l \neq 0 \wedge y = 0 \star x \mapsto l)\end{matrix} \Rrightarrow \begin{matrix}(\exists z. \in T(l).\ l = 0 \wedge y = 1 \star x \mapsto z) \vee \\ (l \neq 0 \wedge y = 0 \star x \mapsto l)\end{matrix}$$

$$(4) : \lambda+1, \mathcal{A}' \models \mathtt{x} = x \star \exists l \in \{0,1\}.\ \mathbf{spin}_r^\lambda(\mathtt{x},l) \star \begin{matrix}(\mathtt{d}=0 \wedge r \Mapsto \blacklozenge) \vee \\ (\mathtt{d}=1 \wedge r \Mapsto (0,1) \wedge l = 0)\end{matrix} \Rrightarrow$$
$$l = 0 \wedge z = 1 \star \mathtt{x} = x \ \star r \Mapsto (l,z) \text{ and}$$
$$\mathcal{A}' \models l = 0 \wedge z = 1 \star \mathtt{x} = x \ \star r \Mapsto (l,z) \text{ stable}$$

Figure 7.5: Side-conditions for lock proof sketch

Remark 7.0.1. Depending on how the lock operation is thought of, it would be reasonable to think of its abstract atomic properties as resulting from the fact that the while loop is constructed to perform precisely one 'step' of progress, where the underlying progress is atomic, rather than the entire while construct. This is a subtle difference, but not quite how the above proof proceeds. Indeed, the primitive atomicity of the CAS operation is actually not necessary for the derivation to hold, only the fact that the region is only updated once. This can be seen in the way the make atomic rule is used before the while rule, and the repeated attempts at CAS are specified with a Hoare triple rather than atomic triple. This subtlety in the proof sketch is exactly the reason why we require an exclusive guard for the make atomic operation. Unfortunately, any modification to the proof system to capture this (i.e. for the derivation to proceed by while, then open region, rather than make atomic, while, update region) would require significant overhaul to provide a mechanism of tracking the quantity of progress made by a while construct and for the while proof rule to be a hybrid triple. It would also require the CAS rule to capture the update to the store and the heap as a single atomic update (I discuss how this could be done in Section 9.2). The complexity and work required to make this extension to while is infeasible and should not be considered as a solution to the exclusivity of the guard required by spinlock.

Remark 7.0.2. TaDA also provides a verification of a specification for spinlock. Unfortunately, it is unclear whether the stability requirement on make atomic is satisfied in that derivation, and the stability requirement on make atomic is not present on the proof rules of that version.

Discussion on how to stack the spinlock verification with more complex clients is in Section 9.2.

Chapter 8

Evaluation

I consider TaDA 2.0 strengths and weaknesses along several axes: ease of use for verification (simplicity of the proof rules, composability and modularity), simplicity and expressivity of the semantic model (which provides more certainty of the soundness proof and improved extensibility) and expressivity of the logic (complexity of verifiable programs).

8.1 In comparison with TaDA

One goal of this project was to produce a logic similar to TaDA's, but precise and sound. This has been achieved - I have to the best of my ability been meticulous with the details of the definitions and lemmas required for the soundness proof of TaDA 2.0, and find additional certainty in the correctness of most of the proof rules by leaning on the equivalences with TaDALive.

- Some of TaDA's proof rules were missing important premises for soundness (some of which covered by a sweeping but incorrect stability assumption deeper in the model) such as make atomic or frame. These problems are corrected in the proof rules of TaDA 2.0. Furthermore, I make consistent the notational choices in TaDA 2.0 with TaDALive, and provide the spinlock example in full detail.
- TaDA's proof rules were not always stated at the correct level of generality or abstraction. For example, the original TaDA consequence rule did not allow for a viewshift of the atomic precondition. TaDA also had in number many more proof rules than TaDA 2.0, as it provided several separate proof rules for atomic specifications, such as a separate frame rule, and several different substitution rules or atomicity rules which are subsumable by the more general rules of TaDA 2.0.
- The semantic model of TaDA is ad hoc. TaDA 2.0 presents a precise and detailed model, which I conjecture is tractable for mechanisation, and has clear abstraction boundaries between different components. How much this improves extensibility to more complex concurrency patterns is difficult to evaluate without doing the further work, but is likely to at least be a significant improvement on the extensibility of TaDA. Indeed, stronger properties such as needing to specify control over environment behaviour at the level of one execution step at a time as done in the TaDALive model requires only an additional axiom on its frame-preserving updates.

There are two places in which TaDA 2.0 is weaker than TaDA. The first is that the semantic model does not explicitly deal in CAP's abstract predicates, which restricts abstraction and thus proof reuse. However, this should not be too difficult to extend the model to handle, and almost all of the necessary components are present in TaDA. The second is in the requirement of the atomic parts of hybrid specifications to not contain program variables. This requirement is how we reached the new proof rules for READ, CAS and FAS, which are strictly weaker than TaDA's (and can be constructed by TaDA's rules). Whether this difference is genuinely a problem is a matter for how the software system should be modelled, as on a hardware level the update to a program variable is not necessarily atomic with the memory update, but it certainly is atomic at the abstraction level of a programs operational semantics and results in a strictly weaker logic.

The stability premises of rules such as make atomic and frame make at first glance TaDA 2.0 appear weaker than TaDA. However, TaDA makes a sweeping stability assumption in the model which is incorrect and encompasses these. Furthermore, without this stability assumption, the rules are unsound, so in fact these premises serve only to provide the sound rules TaDA was trying to express.

I explain how to solve all of these more fully in Section 9.2, and draw attention to where the time constraints of the project made it impossible for me to certain of their correctness.

8.2 In comparison with TaDALive

Another important aim of the project was to reduce the complexity of TaDALive's semantic model. This I have made significant progress with the following changes.

- Removing the PState component of program configurations greatly simplifies the inductive nature of the soundness proof. Remark 6.2.1 discusses how this prevents a typical induction over the structure of TaDALive's PTraces, and how my modification to the operational semantics and change in program configurations resolves this to induct as usual over the length of a trace. A sound inductive principle to handle this additional inductive structure is missing from the TaDALive paper, while I include an explicit example of how this works in TaDA 2.0 in the proof of Theorem 6.2.

- TaDALive's traces are infinite, so most of the inductive proofs in TaDA 2.0 become coinductive in TaDALive.

- TaDALive constructs traces of logical intrumentations corresponding to a concrete trace in the trace safety judgement, while TaDA 2.0 constructs only the initial state and reachable end states. These logical instrumentations are used in TaDALive to check liveness properties, and their absence in TaDA 2.0 eases proof burden when inducting over the trace safety judgement.

- The TaDALive semantic model uses four different types of traces to model the program at different abstraction levels. I maintain the abstraction boundaries in components of the semantic model between logical ghost state versus execution states, while reducing the TaDA 2.0 semantic model to only require one type of execution trace. This greatly reduces the number of interactions between different components of the semantic model and thus make it easier to reason about.

- Removing liveness conditions from the proof rules and simplifying them in certain cases reduces the number of premises from as many as eleven to three - this provides evidence that verifying liveness properties in some cases obscures the safety proof. This process was more than just removing the subjective obligation related components from premises and usually required some back and forth with the soundness proof to settle on the right choices.

These changes motivate the conclusion that TaDA 2.0's semantic model is not simply TaDALive's without liveness, but provides a true reduction in simplicity and indeed the technical work provided in getting the soundness proof of TaDA 2.0 to hold is insufficient for TaDALive.

The complexity of TaDALive's semantic model has left some incorrect technical details unnoticed in the paper, which I have fixed.

- The trace safety judgement is not well-defined, as it asks for the world semantics of assertions without providing a logical store to resolve logical variables in expression evaluations, and similarly, specification semantics do not resolve logical variables when checking the precondition. I explicitly quantify over LStores in the specification semantics and require each LStore to result in a correct trace safety judgement, which has a constant LStore across the whole trace.

- Furthermore, the syntactic sugar for atomic triples results in unsound proof rules as several logical variables are missing from the precondition, and modified variables are unhandled (discussed in Remark 5.1.1 and Section 5.2). This means any derivations of TaDALive proof rules using MUTATE, READ, CAS and FAS are likely to be incorrect. I have resolved this by providing a correct interpretation from atomic triples to hybrid triples and new proof rules for READ, CAS and FAS with the associated soundness proofs. This allows TaDA 2.0 to verifies atomically modified program variables, which TaDALive does not.

Next, we consider the relative expressivity of TaDA 2.0 and TaDALive.

- The focus on liveness properties in TaDALive requires a fairness assumption on the scheduler. By contrast, safety properties have no dependence on fairness, and so TaDA 2.0 could be considered strictly stronger with respect to safety proofs than TaDALive.

- It would not be fair to say that TaDA 2.0 is weaker than TaDALive due to the new, weaker, proof rules for READ, CAS and FAS, as the TaDALive rules are incorrect, and due to the similarity of the other proof rules it would be reasonable to expect TaDA 2.0 to be able to verify safety properties of all of the programs TaDALive can.
- Due to time constraints, I have not been able to produce further, more complex examples of the uses of TaDA 2.0, but it would be surprising if TaDA 2.0 could not verify a CLH lock with the same specification as TaDALive (without the ghost state for liveness). Furthermore, the specification we prove for a spinlock in Chapter 7 is closely aligned with that of TaDALive, so it would be reasonable to infer that it is sufficiently strong to verify fine-grained client code, as done in TaDALive.
- It is reasonable to say that TaDA 2.0 is significantly weaker than TaDALive, as it provides no reasoning for liveness properties. However this was intentional, as liveness is out of the scope of the project, and indeed likely to be in the best interest of further work - the scale and complexity of the TaDALive model makes it harder to extend than TaDA 2.0 to more complex concurerncy patterns. Also, unlike atomicity and ownership based properties which often have be used in tandem to produce complete safety proofs, safety and liveness properties are more separable, so it does not restrict the complexity of programs whose safety properties are verifiable in TaDA 2.0.
- Although unacknowledged in the original TaDALive paper, TaDALive is weaker in modularity than TaDA as it does not support abstract predicates in the semantic model. This is ignored when producing examples such as the spin lock, for which TaDALive does use abstract predicates. In this sense, I have not further weakened the modularity properties of the existing logics by not including abstract predicates in the semantic model of TaDA 2.0.

8.3 In comparison with other work

As was the goal of the project, we have tackled the issue of precision in TaDA and complexity in TaDALive, without yet more complex constructions in the semantic model and proof rules. Therefore, we maintain the relative simplicity of the logic compared to the higher-order Iris, or liveness focused Trillium.

Chapter 9

Conclusion

9.1 Summary

We commenced this project with the aim of producing a precisely stated, sound program logic for ownership and atomicity reasoning about safety properties of programs. To the best of our knowledge, we have achieved this with TaDA 2.0, which is explicit in all of the details, including the necessary side conditions for its proof rules, (and reduces the number by stating them in better generality). In doing so, we have replaced the semantic model of TaDA with a new trace-based model, closely related to that of TaDALive, and then applied a number of simplifications to the model. The most impactful of our simplifications was to remove the PState construct by modifying the operational semantics and program configurations, and reduce our traces to only finite ones, as this permitted a standard induction over traces which is necessary in the soundness proof. We also reduced the number of types of traces of TaDALive model from four to one, greatly reducing the complexity of the model for someone to consider further extensions.

TaDA 2.0 resolves some incorrect technical details in TaDALive, which are a cause for either unsoundnes (in the case of certain primitive proof rules) or incorrect semantics (in the case of the missing LStore in specification semantics). By replacing the syntactic sugar for atomic triples the existing MUTATE rule becomes sound, and the READ, CAS and FAS rules are totally new, intended to handle atomically modified program variables, which TaDALive cannot do.

Then, we provide all of the technical work required to make the soundness proof hold, much of which is omitted from the TaDALive paper and highly nontrivial (as in the case of induction over traces). We also include soundness proofs of proof rules for which a proof exists in TaDALive, so our soundness proof covers strictly more than theirs (the TaDA semantic model is so different that any comparison of complexity or scale of the soundness proof is close to meaningless). We include the soundness proofs of all rules which are new and then a selection more of each type of rule.

We demonstrate through the use of the spinlock example that these new proof rules do not substantially weaken the expressivity of the logic, and furthermore observe that the spin lock verification in TaDA is imprecise and possibly incorrect. As the specification we verify of the spinlock is analogous to that of TaDALive, we conclude that it is strong enough to be used in a range of fine-grained client code.

9.2 Further Work

Exploration of further examples Due to time constraints, I have not been able to produce detailed proofs of the CLH lock verified in TaDALive, nor the client code which makes use of the spin lock verification. A verification of these examples in TaDA 2.0 would strengthen the argument that TaDA 2.0 is at least as expressive as TaDALive with respect to safety properties, and I believe there may be further insight to be made into simplifying the proof rules and semantic model by working through these more complex examples.

TaDALive verifies a blocking counter, using its spinlock to guarantee race-freedom. As the specification TaDA 2.0 provides is equivalent, the construction TaDALive uses should be sufficient to guide a TaDA 2.0 verification of the blocking counter, which is to use two nested regions to abstract away information about the abstract state of the spinlock. I did not have time to investigate whether the removal of liveness components makes it possible to verify the blocking counter with only one shared region, rather than the nesting seen in TaDALive. The requirements on opening these nested regions provide the exclusivity needed by the spinlock to lock and unlock, essentially pushing that reason up to the client, and without

modification to the stability requirement on make atomic cannot be improved upon in the current version of TaDA 2.0.

Requirements on atomic conditions The technical requirement for atomic pre- and post-conditions to not contain any program variables could be potentially be lifted by making some changes to the trace safety judgement. This requirement is imposed shared resources referenced in the atomic precondition must not change in abstract representation due to updates in the program variables outside of the linearisation point. Otherwise the abstract state of the resources would change more than once. Therefore, the trace safety judgement uses a constant $p_a : \mathsf{AVal} \to \mathtt{World}^{\uparrow}_{\mathcal{A}}$ with no dependence on the program store of the trace, enforcing that an update to the shared resources happens only once. This restriction could potentially be lifted by modifying $p_a : \mathsf{PStore} \times \mathsf{AVal} \to \mathtt{World}^{\uparrow}_{\mathcal{A}}$ (the logical variables cannot change in value across a trace so there is no need for an LStore parameter). In order to correctly model the atomicity property, the STUTTER rule would require an additional premise that $p_a(\sigma_0, v) = p_a(\sigma_1, v)$, preventing changes to the program variables breaking atomicity.
I have not implemented this change in TaDA 2.0, as I am uncertain if it may allow race conditions in the event the environment is responsible for completing the linearisation point. The current trace safety judgement marks the linearisation point as the moment in which the local trace observes the update, rather than when the environment makes it. I worry that a race between an environment step completing the linearisation point and the local trace updating a program variable required for the atomic pre- or post-condition could arise from this modification. It may be that the correct way to prevent this is by introducing a new rule to the trace safety judgement for the linearisation point to be marked at the point it is completed by the environment but the subtlety of the trace safety semantics is such that I can't be certain this is correct.
Lifting this restriction would admit the following, stronger, proof rules for READ, CAS and FAS:

$$\textbf{(Read)}\ \frac{}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}n \in \mathbb{Z}.\ \langle\ \mathtt{x} = x \mid \mathtt{E} \mapsto n\ \rangle\ \ \mathtt{x} := [\mathtt{E}]\ \ \langle\ \mathsf{emp} \mid \mathtt{E}[\mathtt{x}/x] \mapsto n \star \mathtt{x} = n\ \rangle}$$

$$\textbf{(CAS)}\ \frac{}{\begin{array}{l}\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}n \in \mathbb{Z}.\langle\ \mathtt{x} = x \mid \mathtt{E}_1 \mapsto n\rangle \\ \qquad\qquad \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3) \\ \qquad\qquad \exists y.\ \left\langle\ \mathsf{emp} \mid \begin{array}{l}(n = \mathtt{E}_2[\mathtt{x}/x] \wedge \mathtt{x} = 1 \star \mathtt{E}_1[\mathtt{x}/x] \mapsto \mathtt{E}_3[\mathtt{x}/x]) \vee \\ (n \neq \mathtt{E}_2[\mathtt{x}/x] \wedge \mathtt{x} = 0 \star \mathtt{E}_1[\mathtt{x}/x] \mapsto n)\end{array}\ \right\rangle\end{array}}$$

$$\textbf{(FAS)}\ \frac{}{\Phi, \lambda, \mathcal{A} \vdash \mathbb{V}n \in \mathbb{Z}.\ \langle\ \mathtt{x} = x \mid \mathtt{E}_1 \mapsto n\rangle\ \ \mathtt{x} := \mathtt{FAS}(\mathtt{E}_1, \mathtt{E}_2)\ \ \langle\ \mathsf{emp} \mid \mathtt{E}_1[\mathtt{x}/x] \mapsto \mathtt{E}_2[\mathtt{x}/x] \star \mathtt{x} = n\ \rangle}$$

These are the rules TaDA provides, and which TaDALive was attempting to capture.

Abstract Predicates TaDA 2.0 does not explicitly support abstract predicates in its semantic model. In line with TaDA, I expect it to be possible to extend the semantic model and proof rule to explicitly deal in abstract predicates, by including a fifth component of worlds corresponding to abstract predicate names and arguments, similar to the existing ρ. An abstract predicate interpretation would be required. Then a step-indexed semantics is given to circumvent circularity problems for soundness in TaDA. Following this approach, it should be feasible to include abstract predicates in TaDA 2.0's worlds, but I am uncertain of the impact this would have on defining interference and frame-preserving updates. Probably the abstract predicate interpretations would have to be *Views*, beyond this I'm unsure of the knock-on effect to the rest of the model.

Further simplifications to the model I have made substantial simplification to the semantic model for TaDA 2.0, but there is still a lot of different components with complex interactions. It is possible that a fresh pair of eyes may be able to further reduce the complexity of the model without reducing its expressivity. As commented on in Remark 5.1.3, the final component of the trace safety judgement, $S \in \mathcal{P}(\mathsf{SState})$ is only used when appending to traces, required in a small number of inductions over the operational semantics. I expect that more time studying the soundness proof would bring rise to proof methods which did not require this component. Furthermore, by imposing the intuitive $\mathcal{A} \models \exists x \in X.P_a(x)$ stable condition on specifications, more simplifications to the soundness proof may be possible - this was suggested to me by the authors of the TaDALive paper.

Stability The stability requirements on FRAME and MAKE ATOMIC are impractically strong, and realistically forbid any interference on certain components beyond that restricted by the environment interference assumption. This is in line with that of TaDALive and does not represent a weakening of TaDA 2.0, as the stability requirements are necessary and a point of unsoundness for TaDA.

(Make atomic)

$$\frac{r \notin dom(\mathcal{A}) \qquad \forall x \in X.\ \mathcal{A} \models \mathbf{t}_r^{\lambda'}(x) \star \lceil G \rceil_r \text{ stable} \qquad \mathcal{A}' = \mathcal{A}[r \mapsto (X, T)] \qquad T \subseteq \mathcal{T}_t(G) \qquad \lambda' < \lambda \quad \Phi, \lambda, \mathcal{A}' \vdash \{\ P_h \star \exists x \in X.\ \mathbf{t}_r^{\lambda'}(x) \star r \Rrightarrow \blacklozenge\ \}\ \mathbb{C}\ \{\exists x, y.\ T(x,y) \star Q_h(x,y)\ \star r \Rrightarrow (x,y)\ \}}{\Phi, \lambda, \mathcal{A} \vdash \forall x \in X. \langle\ P_h \mid \mathbf{t}_r^{\lambda'}(x) \star \lceil G \rceil_r\ \rangle\ \mathbb{C}\ \exists y. \langle\ Q_h(x,y) \mid \mathbf{t}_r^{\lambda'}(y) \star \lceil G \rceil_r \star T(x,y)\ \rangle}$$

Using MAKE ATOMIC as the example, the $\forall x \in X.\ \mathcal{A} \models \mathbf{t}_r^{\lambda'}(x) \star \lceil G \rceil_r$ stable requirement in practice prevents the shared region having any dependence on X. This is the crux of why our spin lock example requires an exclusive guard. An improved logic may be able to further refine the notion of stability or the premises required for these rules to be more permissive, and therefore not require this exclusive guard for the spinlock.

Modifications to proof rules A number of the Hoare proof rules may be stronger than strictly necessary. For example, SEQ, which can only conclude a Hoare triple from the specifications of $\mathbb{C}_1$ and $\mathbb{C}_2$ should be modifiable to accommodate for atomic actions.

$$\textbf{(Seq'.1)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \forall x \in X.\ \langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}_1\ \exists y.\ \langle\ R(x,y) \mid Q_a(x,y)\ \rangle \quad \Phi, \lambda, \mathcal{A} \vdash \{\ R(x,y)\ \}\ \mathbb{C}_2\ \{\ Q_h(x,y)\ \}}{\Phi, \lambda, \mathcal{A} \vdash \forall x \in X.\ \langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}_1\ ;\ \mathbb{C}_2\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}$$

$$\textbf{(Seq'.2)}\ \frac{\Phi, \lambda, \mathcal{A} \vdash \{\ P\ \}\ \mathbb{C}_2\ \{\ R\ \} \qquad \Phi, \lambda, \mathcal{A} \vdash \forall x \in X.\ \langle\ R \mid P_a(x)\ \rangle\ \mathbb{C}_1\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}{\Phi, \lambda, \mathcal{A} \vdash \forall x \in X.\ \langle\ P_h \mid P_a(x)\ \rangle\ \mathbb{C}_1\ ;\ \mathbb{C}_2\ \exists y.\ \langle\ Q_h(x,y) \mid Q_a(x,y)\ \rangle}$$

Specifically, if an atomic action occurs in exactly one of the commands, we should be able to conclude some atomic action of the sequenced command. I believe this should be possible without requiring extension to the semantic model but may require additional premises for soundness.

A similar idea could be applicable to PARALLEL, in which an atomic action occuring in exactly one of the threads can be used to justify an atomic action when viewing the system as both threads. Again, this should be possible without requiring extension to the semantic model, but the complexity of the soundness of PAR is such that it would be harder to certain of its correctness without a mechanization.

Helping One more complex concurrency which some other modern separation logics can verify is *helping*, when the atomic action is completed by some other thread, without necessarily being directly caused by the thread which requires the atomic action. It can even be the case that the thread which takes the atomic action is nondeterministic. An easily digestible example of this pattern is in a ticket lock, where threads wishing to lock the ticketlock will "take a ticket" (increment a next counter), and then wait until some other threads updates the ownership of the lock to match the ticket. The primary reason this is impossible in TaDA (and TaDA 2.0) is that the atomicity rules are designed in such a way that the linearisation point is directly connected to the implementation, and not transferable in any way to other threads. [20] outlines in Chapter 8 how TaDA (and TaDA 2.0) may be extensible to helping, requiring an overhaul of the atomicity tracking resource to be transferable between threads.

Mechanization I believe TaDA 2.0 is precisely stated enough for the definitions and soundness to be mechanizable. This would also allow our examples to be computer verified.

Different memory models The hybrid specifications allow for us to specify the same functional behaviour at different levels of atomicity. Using READ as an example, the following rules have different semantics: the first asserts that the operation happens atomically with the update to the store and heap at the same time, while the second asserts only that the heap operation happens atomically and third simply that it happens.

$$\forall n \in \mathbb{N}.\ \langle \mathtt{x} = x \mid \mathtt{E} \mapsto n \rangle\ \mathtt{x} := [\mathtt{E}]\ \exists y.\ \langle \mathsf{emp} \mid \mathtt{x} = n \star \mathtt{E}[\mathtt{x}/x] \mapsto n \rangle$$
$$\forall n \in \mathbb{N}.\ \langle \mathtt{x} = x \mid \mathtt{E} \mapsto n \rangle\ \mathtt{x} := [\mathtt{E}]\ \exists y.\ \langle \mathtt{x} = n \mid \mathtt{E}[\mathtt{x}/x] \mapsto n \rangle$$
$$\forall n \in \mathbb{N}.\ \langle \mathtt{x} = x \star \mathtt{E} \mapsto n \mid \mathsf{emp} \rangle\ \mathtt{x} := [\mathtt{E}]\ \exists y.\ \langle \mathtt{x} = n \star \mathtt{E}[\mathtt{x}/x] \mapsto n \mid \mathsf{emp} \rangle$$

As discussed earlier in this project, the TaDA 2.0 rules provided for READ, CAS and FAS correspond to the second atomicity abstraction here. The original TaDA paper conjectures that these ideas could be extended for proof systems modelling weaker memory models such as TSO, and to my knowledge has not been explored.

Bibliography

[1] Dinsdale-Young T, Dodds M, Gardner P, Parkinson MJ, Vafeiadis V. Concurrent Abstract Predicates. In: D'Hondt T, editor. ECOOP 2010 - Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010. p. 504-28.

[2] Sergey I. CSL Family Tree;. Available from: https://ilyasergey.net/assets/other/CSL-Family-Tree.pdf.

[3] da Rocha Pinto P, Dinsdale-Young T, Gardner P. TaDA: A Logic for Time and Data Abstraction. In: Jones R, editor. ECOOP 2014 – Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg; 2014. p. 207-31.

[4] Gillian. Verified Software, Department of Computing, Imperial College London; 2023. Accessed: 05/01/2024. Available from: https://vtss.doc.ic.ac.uk/research/gillian.html.

[5] Infer Static Analyzer. Facebook, Inc.; 2023. Accessed: 05/01/2024. Available from: https://fbinfer.com/.

[6] D'Osualdo E, Sutherland J, Farzan A, Gardner P. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. ACM Transactions on Programming Languages and Systems. 2021 nov;43(4):1-134.

[7] O'Hearn P, Reynolds J, Yang H. Local Reasoning about Programs that Alter Data Structures. In: Fribourg L, editor. Computer Science Logic. Berlin, Heidelberg: Springer Berlin Heidelberg; 2001. p. 1-19.

[8] O'Hearn PW. Resources, Concurrency and Local Reasoning. In: CONCUR. vol. 3170 of Lecture Notes in Computer Science. Springer; 2004. p. 49-67.

[9] Owicki S, Gries D. An axiomatic proof technique for parallel programs I. Acta Informatica 6. 1976:319–340.

[10] Brookes S, O'Hearn PW. Concurrent Separation Logic. ACM SIGLOG News. 2016 aug;3(3):47–65. Available from: https://doi.org/10.1145/2984450.2984457.

[11] Jones CB. Specification and Design of (Parallel) Programs. In: IFIP Congress. North-Holland/IFIP; 1983. p. 321-32.

[12] Dodds M, Feng X, Parkinson M, Vafeiadis V. Deny-Guarantee Reasoning. In: Castagna G, editor. Programming Languages and Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. p. 363-77.

[13] Vafeiadis V. Modular fine-grained concurrency verification. University of Cambridge, Computer Laboratory; 2008. UCAM-CL-TR-726. Available from: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf.

[14] Herlihy M, Shavit N. The Art of Multiprocessor Programming, Revised Reprint. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2012.

[15] Herlihy MP, Wing JM. Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems. 1990 jul;12(3):463–492.

[16] Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers. 1979;C-28(9):690-1.

[17] Papadimitriou CH. The Serializability of Concurrent Database Updates. J ACM. 1979 oct;26(4):631–653. Available from: https://doi.org/10.1145/322154.322158.

[18] Calcagno C, Gardner P, Zarfaty U. Local Reasoning about Data Update. Electronic Notes on Theoretical Computer Science. 2007 Apr;172:133-75.

[19] da Rocha Pinto P, Dinsdale-Young T, Gardner P. TaDA: A Logic for Time and Data Abstraction (extended version); 2014.

[20] da Rocha Pinto P. Reasoning with Time and Data Abstractions [phdthesis]. Imperial College London; 2017.

[21] da Rocha Pinto P, Dinsdale-Young T, Gardner P, Sutherland J. Modular Termination Verification for Non-blocking Concurrency. In: Thiemann P, editor. Proceedings of the 25$^{\text{th}}$ European Symposium on Programming (ESOP'16). vol. 9632 of Lecture Notes in Computer Science. Springer; 2016. p. 176-201.

[22] Liang H, Feng X. A program logic for concurrent objects under fair scheduling. SIGPLAN Not. 2016 jan;51(1):385–399. Available from: https://doi.org/10.1145/2914770.2837635.

[23] Parkinson MJ. The Next 700 Separation Logics. In: 2010 Verified Software: Theories, Tools, Experiments. vol. 6217 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 2010. p. 169-82. Available from: https://www.microsoft.com/en-us/research/publication/the-next-700-separation-logics/.

[24] Krebbers R, Jung R, Bizjak A, Jourdan JH, Dreyer D, Birkedal L. The Essence of Higher-Order Concurrent Separation Logic. In: Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings. Berlin, Heidelberg: Springer-Verlag; 2017. p. 696–723. Available from: https://doi.org/10.1007/978-3-662-54434-1_26.

[25] Jung R, Swasey D, Sieczkowski F, Svendsen K, Turon A, Birkedal L, et al. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '15. New York, NY, USA: Association for Computing Machinery; 2015. p. 637–650. Available from: https://doi.org/10.1145/2676726.2676980.

[26] Spies S, Gäher L, Gratzer D, Tassarotti J, Krebbers R, Dreyer D, et al. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021. New York, NY, USA: Association for Computing Machinery; 2021. p. 80–95. Available from: https://doi.org/10.1145/3453483.3454031.

[27] Timany A, Gregersen SO, Stefanesco L, Hinrichsen JK, Gondelman L, Nieto A, et al. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. vol. 8. New York, NY, USA: Association for Computing Machinery; 2024. Available from: https://doi.org/10.1145/3632851.

[28] Dinsdale-Young T, Birkedal L, Gardner P, Parkinson MJ, Yang H. Views: compositional reasoning for concurrent programs. In: POPL. Association for Computing Machinery; 2013. p. 287-300.

Appendix A

Additional Definitions

Notation A.1. Write $f : X \rightharpoonup Y$ for a partial function, and $f : X \rightharpoonup_f Y$ for a finite partial function.

$$
\begin{aligned}
mod(\texttt{skip}) &= \emptyset & mod(\mathbb{C}_1 \ ; \ \mathbb{C}_2) &= mod(\mathbb{C}_1) \cup mod(\mathbb{C}_2) \\
mod(\texttt{x} := \texttt{E}) &= \{\texttt{x}\} & mod(\mathbb{C}_1 \ \ || \ \ \mathbb{C}_2) &= mod(\mathbb{C}_1) \cup mod(\mathbb{C}_2) \\
mod([\texttt{E}_1] := \texttt{E}_2) &= \emptyset & mod(\texttt{var x = E in } \mathbb{C}) &= mod(\mathbb{C}) \setminus \{\texttt{x}\} \\
mod(\texttt{x} := \texttt{CAS}(\texttt{E}_1, \texttt{E}_2, \texttt{E}_3)) &= \{x\} & mod(\texttt{if (B) } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2) &= mod(\mathbb{C}_1) \cup mod(\mathbb{C}_2) \\
mod(\texttt{x} := \texttt{FAS}(\texttt{E}_1, \texttt{E}_2)) &= \{x\} & mod(\texttt{while (B) } \mathbb{C}) &= mod(\mathbb{C}) \\
mod(\texttt{x} := \texttt{new}(\texttt{E})) &= \{x\} & mod(\texttt{y} := f(\vec{\texttt{x}})) &= \{y\} \\
mod(\texttt{dispose}(\texttt{E})) &= \emptyset & mod(\texttt{let } f(\texttt{E}) = \mathbb{C}_1 \texttt{ in } \mathbb{C}_2) &= mod(\mathbb{C}_2)
\end{aligned}
$$

$$
\begin{aligned}
pv_{\texttt{B}}(\texttt{b}) &= \emptyset & pv_{\texttt{B}}(\texttt{v}) &= \{\texttt{v}\} \\
pv_{\texttt{B}}(\neg \texttt{B}) &= pv_{\texttt{B}}(\texttt{B}) & pv_{\texttt{B}}(\texttt{B}_1 \wedge \texttt{B}_2) &= pv_{\texttt{B}}(\texttt{B}_1) \cup pv_{\texttt{B}}(\texttt{B}_2) \\
&\ldots & &\ldots \\
pv_{\texttt{E}}(\texttt{v}) &= \emptyset & pv_{\texttt{E}}(\texttt{E}_1 + \texttt{E}_2) &= pv_{\texttt{E}}(\texttt{E}_1) \cup pv_{\texttt{E}}(\texttt{E}_2) \\
pv_{\texttt{E}}(\texttt{x}) &= \{\texttt{x}\} & pv_{\texttt{E}}(\texttt{E}_1 - \texttt{E}_2) &= pv_{\texttt{E}}(\texttt{E}_1) \cup pv_{\texttt{E}}(\texttt{E}_2) \\
&\ldots & &\ldots
\end{aligned}
$$

$$
\begin{aligned}
pv_{\mathbb{C}}(\texttt{skip}) &= \emptyset & pv_{\mathbb{C}}(\mathbb{C}_1 \ ; \ \mathbb{C}_2) &= pv_{\mathbb{C}}(\mathbb{C}_1) \cup pv_{\mathbb{C}}(\mathbb{C}_2) \\
pv_{\mathbb{C}}(\texttt{x} := \texttt{E}) &= \{x\} \cup pv_{\texttt{E}}(\texttt{E}) & pv_{\mathbb{C}}(\mathbb{C}_1 \ \ || \ \ \mathbb{C}_2) &= pv_{\mathbb{C}}(\mathbb{C}_1) \cup pv_{\mathbb{C}}(\mathbb{C}_2) \\
pv_{\mathbb{C}}([\texttt{E}_1] := \texttt{E}_2) &= pv_{\texttt{E}}(\texttt{E}_1) \cup pv_{\texttt{E}}(\texttt{E}_2) & pv_{\mathbb{C}}(\texttt{var x = E in } \mathbb{C}) &= pv_{\texttt{E}}(\texttt{E}) \cup (pv_{\mathbb{C}}(\mathbb{C}) \setminus \{\texttt{x}\}) \\
pv_{\mathbb{C}}(\texttt{x} := \texttt{CAS}(\texttt{E}_1, \texttt{E}_2, \texttt{E}_3)) &= & pv_{\mathbb{C}}(\texttt{if (B) } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2) &= \\
\{x\} \cup pv_{\texttt{E}}(\texttt{E}_1) \cup pv_{\texttt{E}}(\texttt{E}_2) \cup pv_{\texttt{E}}(\texttt{E}_3) & & pv_{\texttt{B}}(\texttt{B}) \cup pv_{\mathbb{C}}(\mathbb{C}_1) \cup pv_{\mathbb{C}}(\mathbb{C}_2) \\
pv_{\mathbb{C}}(\texttt{x} := \texttt{FAS}(\texttt{E}_1, \texttt{E}_2)) &= \{x\} \cup pv_{\texttt{E}}(\texttt{E}_1) \cup pv_{\texttt{E}}(\texttt{E}_2) & pv_{\mathbb{C}}(\texttt{while (B) } \mathbb{C}) &= pv_{\texttt{B}}(\texttt{B}) \cup pv_{\mathbb{C}}(\mathbb{C}) \\
pv_{\mathbb{C}}(\texttt{x} := \texttt{new}(\texttt{E})) &= \{x\} \cup pv_{\texttt{E}}(\texttt{E}_1) & pv_{\mathbb{C}}(\texttt{y} := f(\texttt{E})) &= \{\texttt{y}\} \cup pv_{\texttt{E}}(\texttt{E}) \\
pv_{\mathbb{C}}(\texttt{dispose}(\texttt{E})) &= pv_{\texttt{E}}(\texttt{E}) & pv_{\mathbb{C}}(\texttt{let } f(\texttt{E}) = \mathbb{C}_1 \texttt{ in } \mathbb{C}_2) &= pv_{\mathbb{C}}(\mathbb{C}_2)
\end{aligned}
$$

Let the right-biased union of two partial functions $\sigma_1 \triangleleft \sigma_2$ be

$$
\sigma_1 \triangleleft \sigma_2(x) = \begin{cases} \sigma_2(x), & x \in dom(h) \\ \sigma_1(x), & \text{otherwise} \end{cases}
$$

Modification to a right biased union is defined to be

$$
(\sigma_1 \triangleleft \sigma_2)[x \mapsto v] = \begin{cases} (\sigma_1[x \mapsto v]) \triangleleft \sigma_2, & x \in dom(\sigma_1) \wedge x \notin dom(\sigma_2) \\ \sigma_1 \triangleleft (\sigma_2[x \mapsto v]), & \text{otherwise} \end{cases}
$$

Definition A.0.1 (Program Expression Evaluation)**.** Define $[\![\cdot]\!]_{\cdot} : (\mathsf{PExp} \uplus \mathsf{PBExp}) \times \mathsf{PStore} \to \mathsf{Val}$

$$
\begin{aligned}
[\![\texttt{true}]\!]_{\sigma_p} &= \text{True} & [\![v]\!]_{\sigma_p} &= v \\
[\![\texttt{false}]\!]_{\sigma_p} &= \text{False} & [\![\texttt{x}]\!]_{\sigma_p} &= \sigma_p(\texttt{x}) \\
[\![\neg \texttt{B}]\!]_{\sigma_p} &= \neg [\![\texttt{B}]\!]_{\sigma_p} & [\![\texttt{E}_1 + \texttt{E}_2]\!]_{\sigma_p} &= [\![\texttt{E}_1]\!]_{\sigma_p} + [\![\texttt{E}_2]\!]_{\sigma_p} \\
[\![\texttt{E}_1 \leq \texttt{E}_2]\!]_{\sigma_p} &= [\![\texttt{E}_1]\!]_{\sigma_p} \leq [\![\texttt{E}_2]\!]_{\sigma_p} & [\![\texttt{E}_1 - \texttt{E}_2]\!]_{\sigma_p} &= [\![\texttt{E}_1]\!]_{\sigma_p} - [\![\texttt{E}_2]\!]_{\sigma_p} \\
&\ldots & &\ldots
\end{aligned}
$$

Definition A.0.2 (Expression Evaluation)**.** Define $[\![\cdot]\!]_{\cdot} : (\mathsf{LExp} \uplus \mathsf{LBExp}) \times \mathsf{Store} \to \mathsf{AVal}$

$$
\begin{aligned}
[\![\texttt{true}]\!]_{\sigma} &= \text{True} & [\![v]\!]_{\sigma} &= v \\
[\![\texttt{false}]\!]_{\sigma} &= \text{False} & [\![\texttt{x}]\!]_{\sigma} &= \sigma(\texttt{x}) \\
[\![\neg B]\!]_{\sigma} &= \neg [\![\texttt{B}]\!]_{\sigma} & [\![x]\!]_{\sigma} &= \sigma(x) \\
[\![E_1 \leq E_2]\!]_{\sigma} &= [\![E_1]\!]_{\sigma} \leq [\![E_2]\!]_{\sigma} & [\![E_1 + E_2]\!]_{\sigma} &= [\![E_1]\!]_{\sigma} + [\![E_2]\!]_{\sigma} \\
&\ldots & [\![E_1 - E_2]\!]_{\sigma} &= [\![E_1]\!]_{\sigma} - [\![E_2]\!]_{\sigma} \\
& & &\ldots
\end{aligned}
$$

Appendix B

Operational Semantics

$$\frac{}{\sigma, h, \texttt{skip} \xrightarrow{\text{loc}}_\varphi \sigma, h, \checkmark} \qquad \frac{}{\sigma, h, \texttt{x} := \texttt{E} \xrightarrow{\text{loc}}_\varphi \sigma[\texttt{x} \mapsto [\![\texttt{E}]\!]_\sigma], h, \checkmark} \qquad \frac{[\![\texttt{E}_1]\!]_\sigma \in dom(h)}{\sigma, h, [\texttt{E}_1] := \texttt{E}_2 \xrightarrow{\text{loc}}_\varphi \sigma, h[[\![\texttt{E}_1]\!]_\sigma \mapsto [\![\texttt{E}_2]\!]_\sigma], \checkmark}$$

$$\frac{[\![\texttt{E}]\!]_\sigma \in dom(h)}{\sigma, h, \texttt{x} := [\texttt{E}] \xrightarrow{\text{loc}}_\varphi \sigma[\texttt{x} \mapsto h([\![\texttt{E}]\!]_\sigma)], h, \checkmark} \qquad \frac{[\![\texttt{E}_1]\!]_\sigma \in dom(h) \qquad h([\![\texttt{E}_1]\!]_\sigma) \neq [\![\texttt{E}_2]\!]_\sigma}{\sigma, h, \texttt{x} := \texttt{CAS}(\texttt{E}_1, \texttt{E}_2, \texttt{E}_3) \xrightarrow{\text{loc}}_\varphi \sigma[\texttt{x} \mapsto 0], h, \checkmark}$$

$$\frac{[\![\texttt{E}_1]\!]_\sigma \in dom(h) \qquad h([\![\texttt{E}_1]\!]_\sigma) = [\![\texttt{E}_2]\!]_\sigma}{\sigma, h, \texttt{x} := \texttt{CAS}(\texttt{E}_1, \texttt{E}_2, \texttt{E}_3) \xrightarrow{\text{loc}}_\varphi \sigma[\texttt{x} \mapsto 1], h[[\![\texttt{E}_1]\!]_\sigma \mapsto [\![\texttt{E}_3]\!]_\sigma], \checkmark}$$

$$\frac{[\![\texttt{E}_1]\!]_\sigma \in dom(h)}{\sigma, h, \texttt{x} := \texttt{FAS}(\texttt{E}_1, \texttt{E}_2) \xrightarrow{\text{loc}}_\varphi, \sigma[\texttt{x} \mapsto h([\![\texttt{E}_1]\!]_\sigma)], h[[\![\texttt{E}_1]\!]_\sigma \mapsto [\![\texttt{E}_2]\!]_\sigma], \checkmark}$$

$$\frac{\begin{array}{c} i = [\![\texttt{E}]\!]_\sigma \qquad i > 0 \qquad \{x, ..., x + i - 1\} \cap dom(h) = \emptyset \\ h' = h[x \mapsto v_0, ..., x + i - 1 \mapsto v_{i-1}] \end{array}}{\sigma, h, \texttt{x} := \texttt{new}(\texttt{E}) \xrightarrow{\text{loc}}_\varphi \sigma[\texttt{x} \mapsto x], h', \checkmark} \qquad \frac{[\![\texttt{E}]\!]_\sigma \in dom(h)}{\sigma, h, \texttt{dispose}(\texttt{E}) \xrightarrow{\text{loc}}_\varphi \sigma, h[[\![\texttt{E}]\!]_\sigma \mapsto \bot], \checkmark}$$

$$\frac{\sigma, h, \mathbb{C}_1 \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}_1'}{\sigma, h, \mathbb{C}_1 \ ; \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}_1' \ ; \ \mathbb{C}_2} \qquad \frac{}{\sigma, h, \checkmark \ ; \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma, h, \mathbb{C}_2} \qquad \frac{\sigma, h, \mathbb{C}_1 \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}_1'}{\sigma, h, \mathbb{C}_1 \ || \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}_1' \ || \ \mathbb{C}_2}$$

$$\frac{\sigma, h, \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}_2'}{\sigma, h, \mathbb{C}_1 \ || \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}_1 \ || \ \mathbb{C}_2'} \qquad \frac{}{\sigma, h, \checkmark \ || \ \checkmark \xrightarrow{\text{loc}}_\varphi \sigma, h, \checkmark} \qquad \frac{}{\sigma, h, \texttt{var x} = \texttt{E in} \ \checkmark \xrightarrow{\text{loc}}_\varphi \sigma, h, \checkmark}$$

$$\frac{\sigma[\texttt{x} \mapsto [\![\texttt{E}]\!]_\sigma], h, \mathbb{C} \xrightarrow{\text{loc}}_\varphi \sigma', h', \mathbb{C}'}{\sigma, h, \texttt{var x} = \texttt{E in} \ \mathbb{C} \xrightarrow{\text{loc}}_\varphi \sigma'[\texttt{x} \mapsto \sigma(\texttt{x})], h', \texttt{var x} = \sigma'(\texttt{x}) \ \texttt{in} \ \mathbb{C}'} \qquad \frac{[\![\texttt{B}]\!]_\sigma}{\sigma, h, \texttt{if (B)} \ \mathbb{C}_1 \ \texttt{else} \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma, h, \mathbb{C}_1}$$

$$\frac{\neg[\![\texttt{B}]\!]_\sigma}{\sigma, h, \texttt{if (B)} \ \mathbb{C}_1 \ \texttt{else} \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma, h, \mathbb{C}_2} \qquad \frac{\neg[\![\texttt{B}]\!]_\sigma}{\sigma, h, \texttt{while (B)} \ \mathbb{C} \xrightarrow{\text{loc}}_\varphi \sigma, h, \checkmark}$$

$$\frac{[\![\texttt{B}]\!]_\sigma}{\sigma, h, \texttt{while (B)} \ \mathbb{C} \xrightarrow{\text{loc}}_\varphi \sigma, h, \mathbb{C} \ ; \ \texttt{while (B)} \ \mathbb{C}} \qquad \frac{\sigma, h, \mathbb{C}_2 \xrightarrow{\text{loc}}_{\varphi'} \sigma', h', \mathbb{C}_2' \qquad \varphi' = \varphi[f \mapsto (\vec{\texttt{x}}, \mathbb{C}_1)]}{\sigma, h, \texttt{let} \ f(\vec{\texttt{x}}) = \mathbb{C}_1 \ \texttt{in} \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \sigma', h', \texttt{let} \ f(\vec{\texttt{x}}) = \mathbb{C}_1 \ \texttt{in} \ \mathbb{C}_2'}$$

$$\frac{}{\sigma, h, \texttt{let} \ f(\vec{\texttt{x}}) = \mathbb{C}_1 \ \texttt{in} \ \checkmark \xrightarrow{\text{loc}}_\varphi \sigma, h, \checkmark}$$

$$\frac{\varphi(f) = (\vec{\texttt{x}}, \mathbb{C})}{\sigma, h, \texttt{y} := f(\vec{\texttt{E}}) \xrightarrow{\text{loc}}_\varphi \sigma, h, \texttt{var ret} = 0 \ \texttt{in} \ (\texttt{var} \ \vec{\texttt{x}} = \vec{\texttt{E}} \ \texttt{in} \ \mathbb{C} \ ; \ \texttt{y} := \texttt{ret})}$$

Figure B.1: Operational Semantics - Local steps (success cases)

$$\frac{[\![\mathtt{E}_1]\!]_\sigma \notin dom(h)}{\sigma, h, [\mathtt{E}_1] := \mathtt{E}_2 \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{[\![\mathtt{E}]\!]_\sigma \notin dom(h)}{\sigma, h, \mathtt{x} := [\mathtt{E}] \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{[\![\mathtt{E}_1]\!]_\sigma \notin dom(h)}{\sigma, h, \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3) \xrightarrow{\text{loc}}_\varphi \lightning}$$

$$\frac{[\![\mathtt{E}_1]\!]_\sigma \notin dom(h)}{\sigma, h, \mathtt{x} := \mathtt{FAS}(\mathtt{E}_1, \mathtt{E}_2) \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{[\![\mathtt{E}]\!]_\sigma \notin dom(h)}{\sigma, h, \mathtt{dispose}(\mathtt{E}) \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{\sigma, h, \mathbb{C}_1 \xrightarrow{\text{loc}}_\varphi \lightning}{\sigma, h, \mathbb{C}_1 \ ; \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \lightning}$$

$$\frac{\sigma, h, \mathbb{C}_1 \xrightarrow{\text{loc}}_\varphi \lightning}{\sigma, h, \mathbb{C}_1 \ \ || \ \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{\sigma, h, \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \lightning}{\sigma, h, \mathbb{C}_1 \ \ || \ \ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{\sigma[\mathtt{x} \mapsto [\![\mathtt{E}]\!]_\sigma], h, \mathbb{C} \xrightarrow{\text{loc}}_\varphi \lightning}{\sigma, h, \mathtt{var\ x} = \mathtt{E\ in}\ \mathbb{C} \xrightarrow{\text{loc}}_\varphi \lightning}$$

$$\frac{\sigma, h, \mathbb{C}_2 \xrightarrow{\text{loc}}_{\varphi'} \lightning \qquad \varphi' = \varphi[f \mapsto (\vec{\mathtt{x}}, \mathbb{C}_1)]}{\sigma, h, \mathtt{let}\ f(\vec{\mathtt{x}}) = \mathbb{C}_1\ \mathtt{in}\ \mathbb{C}_2 \xrightarrow{\text{loc}}_\varphi \lightning} \qquad \frac{f \notin dom(\varphi)}{\sigma, h, \mathtt{y} := f(\vec{\mathtt{E}}) \xrightarrow{\text{loc}}_\varphi \lightning}$$

Figure B.2: Operational Semantics - Local steps (failure cases)

$$\frac{}{c \xrightarrow{\text{env}}_\varphi \lightning} \qquad \frac{}{\sigma, h, \mathbb{C} \xrightarrow{\text{env}}_\varphi \sigma, h', \mathbb{C}}$$

Figure B.3: Operational Semantics - Environment steps

Appendix C

Lemmas

Lemma (4.5 Frame-preserving updates can be augmented with stable worlds)**.** For $\mathcal{A} \in \mathsf{AContext}$, $r, p, q \in \mathtt{World}^{\uparrow}_{\mathcal{A}}, h_0, h_1 \in \mathsf{Heap}$

$$\begin{aligned} &\textbf{if } \mathcal{A} \models r \text{ stable } \textbf{and } (h_0, h_1) \models_{\lambda,\mathcal{A}} p \twoheadrightarrow q \\ &\textbf{then } (h_0, h_1) \models_{\lambda,\mathcal{A}} p * r \twoheadrightarrow q * r \end{aligned}$$

Proof. Take $\mathcal{A} \in \mathsf{AContext}$, $r, p, q \in \mathtt{World}^{\uparrow}_{\mathcal{A}}, h_0, h_1 \in \mathsf{Heap}$ arbitrary and assume $\mathcal{A} \models r$ stable. Take $f \in View_{\mathcal{A}}$ and assume $h_0 \in [\![p * r * f]\!]_\lambda$. As stability is closed under world composition, find that $r * f \in View_{\mathcal{A}}$. Application of the second premise yields $h_1 \in [\![q * r * f]\!]_\lambda$. □

Lemma (5.1 Specifications of skip)**.** Let $\mathbb{S} \in \mathsf{Spec}$ and $\Phi \in \mathsf{FSpec}$. The following holds:

$$\begin{aligned} &\textbf{if } \models_\Phi \mathtt{skip} : \mathbb{S} \\ &\quad \textbf{then } \forall v \in X, \exists v' \in \mathsf{AVal}.\ \lambda, \mathcal{A} \models P_h \star P_a(v) \Rrightarrow Q_h(v, v') \star Q_a(v, v'). \end{aligned}$$

Proof. Take $\mathbb{S}$ arbitrary and assume $\models_\Phi \mathtt{skip} : \mathbb{S}$. Take $\varphi \in \mathsf{FImpl}, \sigma \in \mathsf{Store}, h \in \mathsf{Heap}$ arbitrary such that $\models \varphi : \Phi$ and denote by σ_p, σ_l the PStore and LStore components such that $\sigma = \sigma_p \circ \sigma_l$. It's clear that $(\sigma_p, h, \mathtt{skip})\mathrm{loc}(\sigma_p, h, \checkmark) \in [\![\mathtt{skip}]\!]_\varphi$ from the operational semantics. Take $v \in X, f \in View_{\mathcal{A}}$ arbitrary and assume $h_0 \in [\![\mathcal{W}[\![P_h]\!]^\sigma_{\mathcal{A}} * \mathcal{W}[\![P_a(v)]\!]^{\sigma_l}_{\mathcal{A}} * f]\!]_\lambda$. Thus $h \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$ where $p_h = \mathcal{W}[\![P_h]\!]^{\sigma\circ\sigma_l}_{\mathcal{A}}$ and $p_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star x \in X]\!]^{\sigma_l}_{\mathcal{A}}$ and from $\models_\Phi \mathtt{skip} : \mathbb{S}$ find that $(\sigma, h, \mathtt{skip})\mathrm{loc}(\sigma, h, \checkmark) \in [\![\mathbb{S}]\!]$ and therefore $\exists S.\ (\sigma, h, \mathtt{skip})\mathrm{loc}(\sigma, h, \checkmark) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. Upon examination of the trace safety judgement, find that it must be an application of LinPt which concludes this (as otherwise we cannot meet the condition that linearisation must have happened by termination in the Stutter rule). Therefore from the premises, it must be that as the final Conf concludes in a $\checkmark$, that $(h, h) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow \mathcal{W}[\![Q_h(v, v')]\!]^\sigma_{\mathcal{A}} * \mathcal{W}[\![Q_a(v, v')]\!]^\sigma_{\mathcal{A}}$ and so from $f \in View_{\mathcal{A}}$, conclude $h \in [\![\mathcal{W}[\![Q_h(v, v')]\!]^\sigma_{\mathcal{A}} * \mathcal{W}[\![Q_a(v, v')]\!]^\sigma_{\mathcal{A}} * f]\!]$ and thus we have found some v' such that $\lambda, \mathcal{A} \models P_h \star P_a(v) \Rrightarrow Q_h(v, v') \star Q_a(v, v')$. □

Lemma (6.1 Suffices to consider traces beginning with a local step)**.** Let $\mathbb{C} \in \mathsf{Cmd}, \mathbb{S} \in \mathsf{Spec}$ and $\varphi \in \mathsf{FImpl}$ such that $\models \varphi : \Phi$. The following holds:

$$\begin{aligned} &\textbf{if } \ (\forall\ ((\sigma_0, h_0, \mathbb{C})\ s_0\ \tau) \in [\![\mathbb{C}]\!]_\varphi.\ \textbf{if } s_0 = \mathrm{loc} \textbf{ then } \tau \in [\![\mathbb{S}]\!]) \\ &\quad \textbf{then } (\forall \tau \in [\![\mathbb{C}]\!]_\varphi.\ \tau \in [\![\mathbb{S}]\!]) \end{aligned}$$

Proof. Take $\mathbb{C}, \mathbb{S}, \varphi$ arbitrary and assume $\forall \tau \in [\![\mathbb{C}]\!]_\varphi.\ s_0 = \mathrm{loc} \implies \tau \in [\![\mathbb{S}]\!]$. Take $\tau \in [\![\mathbb{C}]\!]_\varphi$ arbitrary. We induct on the number of initial environment steps. Obviously if the first step of τ is a local step, then the result is given by the assumption. Let's assume that for n initial environment steps the result holds. Take $\tau \in [\![\mathbb{C}]\!]_\varphi$ arbitrarily except that the first $n+1$ steps must be environment steps and the $n+2$'th step must be a local step. Take σ_l, v arbitrary and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$, where h_0 is the first heap of τ and p_h, p_a are as in the definition of $[\![\mathbb{S}]\!]$. We need to show that $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. By construction $\tau = (\sigma, h_0, \mathbb{C})\mathrm{env}(\sigma, h_1, \mathbb{C})\tau'$, where τ' begins with no more than n environment steps, so $\tau' \in [\![\mathbb{S}]\!]$. We aim to apply Env and the inductive hypothesis to give us the result. In particular, we need to check that $\forall v' \in X, .\ E(v') \implies \exists S'_v.\ (\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S_{v'}$. So take $v' \in X$ arbitrary and assume $E(v')$, that is, that $\exists p_e, p'_e$ such that

$$h_0 \in [\![p_h * p_a(v) * p_e]\!] \tag{C.1}$$

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p'_e \tag{C.2}$$

From C.1, C.2, and $p_h \in View_{\mathcal{A}}$, deduce that $h_1 \in [\![p_h * p_a(v') * p'_e]\!]_\lambda \subseteq [\![p_h * p_a(v') * \text{True}_{\mathcal{A}}]\!]_\lambda$ and so from $(\sigma, h_1, \mathbb{C})\tau' \in [\![\mathbb{S}]\!]$ and the stores not changing in environment steps, we find some $S_{v'}$ such that $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S_{v'}$. Finally, we have satisfies the premises of Env, and so we can find S such that $\tau \text{env} C \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. □

Lemma (6.2 Trace safety is closed under appending env steps)**.** Let $\mathbb{S} \in \mathsf{Spec}$, τ env $C \in \mathsf{Trace}$, $\sigma_l \in \mathsf{LStore}$ and $(p_h, p_a, v) \in \mathsf{SState}$. The following holds:

$$\begin{aligned} &\textbf{if } \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S \\ &\quad \textbf{then } \exists S'.\ \tau \text{ env } C \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S' \end{aligned}$$

Proof. Take $\mathbb{S}, \tau \text{env} C \in \mathsf{Trace}, \sigma_l, (p_h, p_a, v)$ arbitrary and assume $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. We aim to show $\exists S'.\ \tau \text{env} C \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S'$ by induction on the structure of the trace safety judgement.

- **Case: Term** Assume $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ by an application of Term, i.e. $\tau = (\sigma, h, \mathbb{C})$, and $S = \{(p_h, p_a, v)\}$. If $C = \lightning$, an application of Env$\lightning$ finds $(\sigma, h, \mathbb{C})\text{env}\lightning \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \emptyset$. If $C = (\sigma, h', \mathbb{C})$, then Term finds $\forall v' \in X.\ (\sigma, h', \mathbb{C}) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), \{(p_h, p_a, v')\}$, and then Env or Env' finds $(\sigma, h, \mathbb{C})\text{env}(\sigma, h', \mathbb{C}) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$.

- **Case: Stutter, LinPt** These cases are straightforward application of the inductive hypothesis - I do the Stutter case explicitly. Assume $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ by an application of Stutter, i.e. $\tau = (\sigma_1, h_1, \mathbb{C}_1)\text{loc}(\sigma_2, h_2, \mathbb{C}_2)\tau'$. From the premises of Stutter, find that $(h_1, h_2) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * p_a(v)$, $\mathbb{C}_2 = \checkmark \implies p'_h = \mathcal{W}[\![Q_h(v)]\!]_{\mathcal{A}}^{\sigma_2 \circ \sigma_l} \wedge v \in \mathsf{AVal} \times \mathsf{AVal}$, and $(\sigma_2, h_2, \mathbb{C}_2)\tau' \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S$. From the inductive hypothesis, find that $\exists S'.\ (\sigma_2, h_2, \mathbb{C}_2)\tau' \text{env} C \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S'$ and from Stutter we can stitch that straight back up to find for the same S' that $\tau \text{env} C \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S'$.

- **Case: Env, Env'** These are also straightforward application of the inductive hypothesis - I do the Env' case. Assume $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, \langle v, v' \rangle), S$ by an application of Env', i.e. $\tau = (\sigma, h_1, \mathbb{C})\text{env}(\sigma, h_2, \mathbb{C})\tau'$. From the premises of Env', find that if $\exists p_e, p'_e$ such that $h_1 \in [\![p_h * p_e]\!]_\lambda \wedge (h_1, h_2) \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e$ then $(\sigma, h_2, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, \langle v, v' \rangle), S$ and otherwise $S = \emptyset$. Apply the inductive hypothesis to the first case to find $\exists S'.\ (\sigma, h_2, \mathbb{C})\tau' \text{env} C \models_{\mathbb{S}} \sigma_l, (p_h, p_a, \langle v, v' \rangle), S'$ and then reapply Env' to find $\exists S'.\ \tau \text{env} C \models_{\mathbb{S}} (p_h, p_a, \langle v, v' \rangle), S'$.

- **Case: Env$\lightning$** Assume $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ by an application of Env$\lightning$, i.e. $\tau = (\sigma, h, \mathbb{C})\text{env}\lightning\tau'$. Then $(\sigma, h, \mathbb{C})\text{env}\lightning\tau' \text{env} C \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \emptyset$ also by an application of Env$\lightning$.

□

Lemma (6.3 Semantics of sequenced commands)**.** For $\mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}, \varphi \in \mathsf{FImpl}$,

$$[\![\mathbb{C}_1 \ ; \ \mathbb{C}_2]\!]_\varphi = [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$$

Proof. Take $\mathbb{C}_1, \mathbb{C}_2, \varphi$ arbitrary. It's clear that every $\tau \in [\![\mathbb{C}_2]\!]_\varphi$ begins with $(\_, \_\mathbb{C}_2)$ so the sequencing is well-defined. I prove both inclusions.
$[\![\mathbb{C}_1 \ ; \ \mathbb{C}_2]\!]_\varphi \subseteq [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$. Induct on $\tau \in [\![\mathbb{C}_1 \ ; \ \mathbb{C}_2]\!]_\varphi$. There are 2 relevant rules from the operational semantics:

$$\frac{(\sigma, h, \mathbb{C}_1) \xrightarrow{\text{loc}}_\varphi (\sigma, h, \mathbb{C}'_1)}{(\sigma, h, \mathbb{C}_1 \ ; \ \mathbb{C}_2) \xrightarrow{\text{loc}}_\varphi (\sigma, h, \mathbb{C}'_1 \ ; \ \mathbb{C}_2)} \ (1) \qquad \frac{}{(\sigma, h, \checkmark \ ; \ \mathbb{C}_2) \xrightarrow{\text{loc}}_\varphi (\sigma, h, \mathbb{C}_2)} \ (2)$$

Let's begin with some $\tau = (\sigma, h, \mathbb{C}_1 \ ; \ \mathbb{C}_2)$, i.e. no steps have been taken. Then for $\tau' = (\sigma, h, \mathbb{C}_1) \in [\![\mathbb{C}_1]\!]_\varphi$, $\tau = \tau'_{;\mathbb{C}_2} \in [\![\mathbb{C}_1]\!]_{\varphi;\mathbb{C}_2} \subseteq [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$. Now consider $\tau \in [\![\mathbb{C}_1]\!]_\varphi$ of length 1, i.e. rule (1) has been used. Then $\tau = (\sigma, h, \mathbb{C}_1 \ ; \ \mathbb{C}_2)\text{loc}(\sigma, h, \mathbb{C}_1 \ ; \ \mathbb{C}_2)$, where $\tau' = (\sigma, h, \mathbb{C}_1)\text{loc}(\sigma, h, \mathbb{C}'_1) \in \mathsf{Trace}_\varphi$. Then clearly $\tau = \tau'_{;\mathbb{C}_2}$, so $\tau \in [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$.
Now consider $\tau s_i(\sigma_i, h_i, \mathbb{C}_i)$ where the final step is justified by (2). From the induction hypothesis, we find $\tau \in [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$ and that in both cases $\tau \in [\![\mathbb{C}_1]\!]_{\varphi;\mathbb{C}_2}$, i.e. $\exists \tau' \in [\![\mathbb{C}_1]\!]_\varphi.\ \tau = \tau'_{;\mathbb{C}_2}$. In the (2) case, we find that $\mathbb{C}_i = \mathbb{C}'_i \ ; \ \mathbb{C}_2$, and therefore from the premise of (2) that $\tau s_i(\sigma_i, h_i, \mathbb{C}_i) = (\tau' s_i(\sigma, h, \mathbb{C}'_i))_{;\mathbb{C}_2} \in [\![\mathbb{C}_1]\!]_{\varphi;\mathbb{C}_2} \subseteq [\![\mathbb{C}_1]\!]_\varphi ; [\![\mathbb{C}_2]\!]_\varphi$.
There are two more cases to consider: $\tau s_i(\sigma_i, h_i, \mathbb{C}_i)$, where $\mathbb{C}_i$ is no longer a sequencing construct, i.e. $\mathbb{C}_1$ has terminated, and environment steps. In the first, τ is divisible into $\tau_1 \in [\![\mathbb{C}_1]\!]_{\varphi;\mathbb{C}_2}$ and

$\tau_2 \in [\![\mathbb{C}_2]\!]_\varphi$ such that $\tau = \tau_1;\tau_2$, and it's clear therefore that $\tau_2 s_i(\sigma_i, h_i, \mathbb{C}_i) \in [\![\mathbb{C}_2]\!]_\varphi$ so we are done. For environment steps, again it's clear that if $\tau \mathrm{env}(\sigma_i, h_i, \mathbb{C}_i)$, then this final step can be justifiably sequenced onto $\tau \in [\![\mathbb{C}_1]\!]_\varphi; [\![\mathbb{C}_2]\!]_\varphi$ regardless of which case is used to justify membership.

$[\![\mathbb{C}_1]\!]_\varphi; [\![\mathbb{C}_2]\!]_\varphi \subseteq [\![\mathbb{C}_1 \ ; \ \mathbb{C}_2]\!]_\varphi$ direction is obvious. □

Lemma (6.4 Terminated traces satisfying precondition satisfy postcondition)**.** Let $\tau \in \mathsf{Trace}$, $\mathbb{S} \in \mathsf{Spec}$, $\sigma_l \in \mathsf{LStore}$, $p_h \in View_{\mathcal{A}}$ and $v \in X$. Then, for $p_a = \lambda x.\ \mathcal{W}[\![P_a(x)]\!]^{\sigma_l}_{\mathcal{A}}$ and $q_h = \mathcal{W}[\![Q_h]\!]^{\sigma_{i+1} \circ \sigma_l}_{\mathcal{A}}$, following holds:

$$\textbf{if} \ \ h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda \ \textbf{and} \ \exists S.\ (S \text{ is nonempty } \textbf{and } \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S) \ \textbf{and} \ \tau = \tau'(h_i, \sigma_i, \mathbb{C}_i) s_i (h_{i+1}, \sigma_{i+1}, \checkmark) \ \textbf{then} \ h_{i+1} \in [\![q_h * \mathrm{True}_{\mathcal{A}}]\!]_\lambda.$$

Proof. Take $\tau, \mathbb{S}, \sigma_l, v$ arbitrary and assume $\tau = \tau'(h_i, \sigma_i, \mathbb{C}_i) s_i (h_{i+1}, \sigma_{i+1}, \checkmark)$, $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ and $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$. Proceed by induction on the length of τ':
Consider $\tau' = (\sigma_0, h_0, \mathbb{C}_0)$ a single state. From our assumptions, $\exists S.\ \tau \models \sigma_l, (p_h, p_a, v), S$ and by definition of the trace safety judgement, it must be that $S = \{(p'_h, p_a, v)\}$, for some p'_h. Regardless of whether Stutter or LinPt is used to get the safety result, the premise of the rule is that $(h_i, h_{i+1}) \models_{\lambda,\mathcal{A}} p_h * \mathtt{Emp}_{\mathcal{A}} \twoheadrightarrow p'_h * \mathtt{Emp}_{\mathcal{A}}$ and therefore from the initial assumption on h_0 we find that $h_{i+1} \in [\![p'_h * \mathrm{True}_{\mathcal{A}}]\!]$. As τ ends with $\checkmark$, that $p'_h = q_h$.
So let $\tau' = (\sigma_0, h_0, \mathbb{C}) s_0 (\sigma_1, h_1, \mathbb{C}_1) \tau''$, where the result holds for $(\sigma_1, h_1, \mathbb{C}_1)\tau'' s_i (h_{i+1}, \sigma_{i+1}, \checkmark)$. If $s_0 = \mathrm{env}$, $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ is either an application of Env or Env', in either case, the premises of the proof rule and S nonempty find $(\sigma_1, h_1, \mathbb{C}_1)\tau'' s_i (h_{i+1}, \sigma_{i+1}, \checkmark) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S$ in precisely the cases for which $\exists p_e, p'_e.\ h_0 \in [\![p_h * p_a(v) * p_e]\!] \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p_e$ in the case of Env (and there exists such a case), or $v = v'$ and $\exists p_e, p'_e.\ h_0 \in [\![p_h * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p_e$ in the case of Env'. In either, we find the necessary premises to apply the inductive hypothesis and find $h_{i+1} \in [\![q_h * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$. If $s_0 = \mathrm{loc}$, then the safety judgement is by an application of either Stutter or LinPt, and similarly to in the τ'' is a single state case, the frame-preserving update premise and the assumption on h_0 provide exactly what we need to apply the inductive hypothesis and find $h_{i+1} \in [\![q_h * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$. □

Lemma (6.5 Safety of concatenated traces)**.** For $\mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}, P, Q, R \in \mathsf{Assert}, \varphi \in \mathsf{FImpl}$ such that $\models \varphi : \Phi, \tau \in [\![\mathbb{C}_1]\!]_\varphi; [\![\mathbb{C}_2]\!]_\varphi$, **if** $\models_\Phi \mathbb{C}_1 : \mathbb{S}_1$ **and** $\models_\Phi \mathbb{C}_2 : \mathbb{S}_2$, where

$$\mathbb{S}_1 = \mathbb{V} x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle \ \cdot \ \exists y.\ \langle\ R \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$
$$\mathbb{S}_2 = \mathbb{V} x \in \mathsf{AVal}.\ \langle\ R \mid \mathsf{emp}\ \rangle \ \cdot \ \exists y.\ \langle\ Q \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

then $\tau \in [\![\mathbb{S}]\!]$, where

$$\mathbb{S} = \mathbb{V} x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle \ \cdot \ \exists y.\ \langle\ Q \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

Proof. Take $\mathbb{C}_1, \mathbb{C}_2, P, Q, R, \varphi, \tau \in [\![\mathbb{C}_1]\!]_\varphi; [\![\mathbb{C}_2]\!]_\varphi$ arbitrary and assume $\models_\Phi \mathbb{C}_1 : \mathbb{S}_1$, $\models_\Phi \mathbb{C}_2 : \mathbb{S}_2$. In the case $\tau \in [\![\mathbb{C}_1]\!]_{;\mathbb{C}_2}$, the assumption $\models_\Phi \mathbb{C}_1 : \mathbb{S}_1$ directly gives some S such that $\tau \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v), S$. As $\mathbb{S}$ and $\mathbb{S}_1$ differ only in their postcondition, an induction on the trace safety judgement allows us to immediately conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ (we know τ does not terminate which is the only place we use the Hoare postcondition in the trace safety judgement).

So let's consider the case $\tau \in \{\ \tau_1;\tau_2 \mid \tau_1 \in \mathbb{T}_1, \tau_2 \in \mathbb{T}_2, \tau_1;\tau_2 \neq \bot\ \}$, i.e. find some $\tau_1 \in [\![\mathbb{C}_1]\!]_\varphi$, $\tau_2 \in [\![\mathbb{C}_2]\!]_\varphi$ such that $\tau = \tau_1;\tau_2$. Proceed by induction on the length of τ_1. Take σ_l, v arbitrary and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$ for $p_h = \mathcal{W}[\![P]\!]^{\sigma_0 \circ \sigma_l}_{\mathcal{A}}$ and $p_a = \lambda x.\ \mathtt{Emp}_{\mathcal{A}}$. We need to show $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$.
τ_1 must have a local step, as $\tau_1 \in [\![\mathbb{C}_1]\!]_\varphi$ and terminates.
For $\tau_1 = (\sigma_0, h_0, \mathbb{C}_1)\mathrm{loc}(\sigma_1, h_1, \checkmark)$, from the hypotheses find $\tau_1 \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v), S_1$, by LinPt and then Term, with $S_1 = \{(r_h, p_a, \langle v, v\rangle)\}$. The premises of LinPt, the assumption on h_0 and Lemma 6.4 tell us that $h_1 \in [\![r_h * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$ where $r_h = \mathcal{W}[\![R]\!]^{\sigma_i \circ \sigma_l}_{\mathcal{A}}$. Again from the sequencing connective, we know τ_2's initial state is $(\sigma_1, h_1, \mathbb{C}_2)$. We have just verified that h_1 satisfies the precondition of $\mathbb{S}_2$, so from $\models_\Phi \mathbb{C}_2 : \mathbb{S}_2$, find some S_2 such that $\tau_2 \models_{\mathbb{S}_2} \sigma_l, (r_h, p_a, v), S_2$. As $\mathbb{S}$ and $\mathbb{S}_2$ differ only in the precondition, find that $\tau_2 \models_{\mathbb{S}} \sigma_l, (r_h, p_a, v'), S_2$. We can use Stutter to check $(\sigma_1, h_1, \checkmark; \mathbb{C}_2)\mathrm{loc}\tau_2 \models_{\mathbb{S}} \sigma_l, (r_h, p_a, v) S_2$: as this step makes no modification to the store or heap (by the operational semantics), the premises of Stutter are trivially satisfied. The premises of LinPt on τ_1 suffice to apply Stutter a second time and find $(\sigma_0, h_0, \mathbb{C}_1; \mathbb{C}_2)\mathrm{loc}(\sigma_1, h_1, \checkmark; \mathbb{C}_2)\tau_2 = \tau_1;\tau_2 \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S_2$.
For $\tau_1 = (\sigma_0, h_0, \mathbb{C}_1)\mathrm{loc}(\sigma_1, h_1, \checkmark)\tau'$ with τ' only having environment steps, we induct on the number of environment steps to show $\exists S.\ (\sigma_1, h_1, \checkmark)\tau'; \tau_2 \models_{\mathbb{S}} \sigma_l, (r_h, p_a, v), S$. The case of τ' with no steps is by application of Stutter to $\tau_2 \models_{\mathbb{S}} \sigma_l, (r_h, p_a, v), S$ to give $(\sigma_1, h_1, \checkmark \ \ || \ \ \mathbb{C}_2)\mathrm{loc}\tau_2 \models_{\mathbb{S}} \sigma_l, (r_h, p_a, v), S$,

as from the definition of the sequencing operator we have the heap does not change in this step, so $(h_1, h_1) \models_{\lambda,\mathcal{A}} r_h * p_a(v) \twoheadrightarrow r_h * p_a(v)$, and clearly this step is not terminating. For $(\sigma_1, h_1, \checkmark)\tau' = (\sigma_1, h_1, \checkmark)\mathrm{env}(\sigma_1, h_2, \checkmark)\tau''$, the inductive hypothesis gives us $\forall v', \exists S.\ (\sigma_1, h_2, \checkmark)\tau'' \models_{\mathbb{S}} (r_h, p_a, v'), S$. Immediately, we can apply Env to yield $\exists S'.\ (\sigma_1, h_1, \checkmark)\mathrm{env}(\sigma_1, h_2, \checkmark)\tau'' \models_{\mathbb{S}} (r_h, p_a, v'), S'$. Finally, as in the case with no env steps, an application of Stutter will allow us to deduce
$(\sigma_0, h_0, \mathbb{C}_1)\mathrm{loc}(\sigma_1, h_1, \checkmark)\mathrm{env}(\sigma_1, h_2, \checkmark)\tau'' \models_{\mathbb{S}} (p_h, p_a, v), S'$ with the premises immediately transferable from the construction of $(\sigma_0, h_0, \mathbb{C}_1)\mathrm{loc}(\sigma_0, h_0, \checkmark) \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v), S'$.

For $\tau_1 = (\sigma_0, h_0, \mathbb{C}_1)\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1')\tau'$, find $\tau_1 \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v), S_1$ is by either Stutter or LinPt. From the premises and the inductive hypothesis, $(\sigma_1, h_1, \mathbb{C}_1')\tau'; \tau_2 \models_{\mathbb{S}} \sigma_l, (p_h', p_a, v), S_2$. As $\mathbb{C}_1'; \mathbb{C}_2$ cannot be $\checkmark$ and $p_a(v) = \mathcal{W}[\![Q_a(v, v)]\!]^{\sigma_l}_{\mathcal{A}}$, the premises of Stutter are immediately applicable to find $\tau_1; \tau_2 \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S_2$.

For $\tau_1 = (\sigma_0, h_0, \mathbb{C}_1)\mathrm{env}(\sigma_0, h_1, \mathbb{C}_1)\tau'$, find $\tau_1 \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v), S_1$ is by either Env or Env'. In the Env case, the premises tell us for $E(v') \triangleq h_0 \in [\![p_h * p_a(v) * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p_e'$ that $E(v') \implies ((\sigma_0, h_1, \mathbb{C}_1)\tau') \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v'), S_{v'}$. For each v' such that $E(v')$, the inductive hypothesis tells us that $\exists S_{v'}.\ ((\sigma_0, h_1, \mathbb{C}_1)\tau'); \tau_2 \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S_{v'}$ thus we can reapply Env to find $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$.
In the Env' case the premises tell us that if $\exists p_e, p_e'$ such that $h_0 \in [\![p_h * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p_e'$ then $(\sigma_0, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}_1} \sigma_l, (p_h, p_a, v), S_1$ else $S_1 = \emptyset$. As $p_a = \lambda x.\ \mathtt{Emp}_{\mathcal{A}}$, this is precisely equivalent to $E(v') \triangleq h_0 \in [\![p_h * p_a(v) * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p_e'$. So we find from the inductive hypothesis that $E(v') \implies ((\sigma_0, h_1, \mathbb{C}_1)\tau'); \tau_2 \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S_{v'}$ (and that either $\forall v'.\ E(v')$ or $\forall v'.\ \neg E(v')$). Application of Env gives us $\exists S.\ \tau_1; \tau_2 \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. □

Lemma (6.6)**.** For $\varphi \in \mathsf{FImpl}, \mathbb{C} \in \mathsf{Cmd}, \tau \in \mathsf{Trace}$,

$$\tau(\sigma_0, h_0, \mathbb{C}_0)s_0(\sigma_1, h_1, \mathbb{C}_1) \in [\![\mathbb{C}]\!]_\varphi \implies \forall \mathtt{x} \in \mathsf{PVar} \setminus \mathit{mod}(\mathbb{C}).\ \sigma_0(\mathtt{x}) = \sigma_1(\mathtt{x})$$

Proof. Take $\varphi, \mathbb{C}, \tau$ arbitrary and assume $\tau(\sigma_0, h_0, \mathbb{C}_0)s_0(\sigma_1, h_1, \mathbb{C}_1) \in [\![\mathbb{C}]\!]_\varphi$. Take $\mathtt{x} \in \mathsf{PVar} \setminus \mathit{mod}(\mathbb{C})$ arbitrary and proceed by induction on the operational semantics. Of the base cases, there are 5 of which change the store (assignment, read, CAS, FAS and alloc), for which the only variables changed are those in $\mathit{mod}(\mathbb{C})$. Of the inductive cases, all are trivial application of the inductive hypothesis except for the var rule. In the case of var, the variables which may change from σ_0 to σ_1 is a subset of $\{\ \mathtt{y} \mid \mathtt{y} \in \mathit{mod}(\mathbb{C}') \wedge \mathtt{y} \neq \mathtt{x}\ \}$ of the inner command, but given the definition of $\mathit{mod}(\mathtt{var\ x = E\ in}\ \mathbb{C}')$, this implies that $\mathtt{x}$ for which $\sigma_0(\mathtt{x}) \neq \sigma_1(\mathtt{x})$ is such that $\mathtt{x} \in \mathit{mod}(\mathbb{C})$. □

Lemma (6.7 Semantics of parallel commands is bowtie of semantics of each command)**.**

$$\forall \mathbb{C}_1, \mathbb{C}_2 \in \mathsf{Cmd}, \varphi \in \mathsf{FImpl}.\ [\![\mathbb{C}_1 \ \|\ \mathbb{C}_2]\!]_\varphi \subseteq [\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_2]\!]_\varphi$$

Proof. Take $\tau \in [\![\mathbb{C}_1 \ \|\ \mathbb{C}_2]\!]_\varphi$ arbitrary. We need to find τ_1, τ_2 such that $\tau = \tau_1 \bowtie \tau_2$. We proceed by induction on the length of τ.
For τ of only one configuration, it's clear from $[\![\mathbb{C}_1 \ \|\ \mathbb{C}_2]\!]_\varphi$ that $\tau = (\sigma, h, \mathbb{C}_1 \ \|\ \mathbb{C}_2)$. Then $(\sigma, h, \mathbb{C}_1)$ and $(\sigma, h, \mathbb{C}_2)$ are in the semantics of $\mathbb{C}_1$ and $\mathbb{C}_2$ respectively, and that $(\sigma, h, \mathbb{C}_1) \bowtie (\sigma, h, \mathbb{C}_2) = \tau$.
For the inductive cases, consider separately when environment steps versus local steps are appended. Begin with environment steps: take $\tau = \tau'(\sigma, h_1, \mathbb{C}')\mathrm{env}(\sigma, h_2, \mathbb{C}')$. From the inductive hypothesis, find τ_1, τ_2 such that $\tau'(\sigma, h_1, \mathbb{C}') = \tau_1 \bowtie \tau_2$. From the definition of $\bowtie$, find that $\tau_1[-1] = (\sigma, h_1, C_1)$, $\tau_2[-1] = (\sigma, h_1, C_2)$ for some $C_1, C_2 \in \mathsf{Cmd}$ such that $\mathbb{C}' = C_1 \ \|\ C_2$. Then it must be that $\tau_1\mathrm{env}(\sigma, h_2, C_1) \bowtie \tau_2\mathrm{env}(\sigma, h_2, C_2) = \tau$.
For local steps, take $\tau = \tau'(\sigma_1, h_1, \mathbb{C}')\mathrm{loc}(\sigma_2, h_2, \mathbb{C}'')$. From Lemma 6.6 and the well-formedness condition $\mathit{mod}(\mathbb{C}_1 \ \|\ \mathbb{C}_2) = \emptyset$, we have that $\sigma_1 = \sigma_2$. Upon analysis of the operational semantics, it's clear the relevant cases are:

$$\frac{\sigma, h, \mathbb{C}_1 \xrightarrow{\mathrm{loc}}_\varphi \sigma', h', \mathbb{C}_1'}{\sigma, h, \mathbb{C}_1 \ \|\ \mathbb{C}_2 \xrightarrow{\mathrm{loc}}_\varphi \sigma', h', \mathbb{C}_1' \ \|\ \mathbb{C}_2}\ (1) \qquad \frac{\sigma, h, \mathbb{C}_2 \xrightarrow{\mathrm{loc}}_\varphi \sigma', h', \mathbb{C}_2'}{\sigma, h, \mathbb{C}_1 \ \|\ \mathbb{C}_2 \xrightarrow{\mathrm{loc}}_\varphi \sigma', h', \mathbb{C}_1 \ \|\ \mathbb{C}_2'}\ (2)$$

$$\frac{}{\sigma, h, \checkmark \ \|\ \checkmark \xrightarrow{\mathrm{loc}}_\varphi \sigma, h, \checkmark}\ (3)$$

I do cases (1) and (3) - case (2) is identical to (1). From the inductive hypothesis, find τ_1, τ_2 such that $\tau'(\sigma_1, h_1, \mathbb{C}') = \tau_1 \bowtie \tau_2$. From the definition of $\bowtie$, find that $\tau_1[-1] = (\sigma_1, h_1, C_1)$, $\tau_2[-1] = (\sigma_1, h_1, C_2)$ for some $C_1, C_2 \in \mathsf{Cmd}$ such that $\mathbb{C}' = C_1 \ \|\ C_2$. In the case τ is constructed using (1), see that

$\tau_1 \mathrm{loc}(\sigma_2, h_2, C_1') \in [\![\mathbb{C}_1]\!]_\varphi$, from the premise of (1) and $\tau_1 \in [\![\mathbb{C}_1]\!]_\varphi$, and see that $\mathbb{C}'' = C_1' \;||\; C_2$. Clearly $\tau_2 \mathrm{env}(\sigma_2, h_2, C_2) \in [\![\mathbb{C}_2]\!]_\varphi$ as $\tau_2 \in [\![\mathbb{C}_2]\!]_\varphi$ and the final step is justified by an env step in the operational semantics (as $\sigma_1 = \sigma_2$). By definition of $\bowtie$, $\tau_1 \mathrm{loc}(\sigma_2, h_2, C_1') \bowtie \tau_2 \mathrm{env}(\sigma_2, h_2, C_2) = \tau$, so $\tau \in [\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_2]\!]_\varphi$.
Finally, we consider $\tau = \tau'(\sigma, h, \checkmark \;||\; \checkmark)\mathrm{loc}(\sigma, h, \checkmark)$. From the inductive hypothesis, find τ_1, τ_2 as usual such that $\tau'(\sigma, h, \checkmark \;||\; \checkmark) = \tau_1 \bowtie \tau_2$ and observe from the definition of $\bowtie$ that $\tau_i[-1] = (\_, \_, \checkmark)$ for $i = 1, 2$, and conclude that $\tau \in [\![\mathbb{C}_1]\!]_\varphi \bowtie [\![\mathbb{C}_2]\!]_\varphi$ from the second construct of the definition of $\bowtie$ lifted to sets. □

Lemma (6.8 Safety of parallel traces)**.** For $\tau_1, \tau_2 \in \mathsf{Trace}, v \in \mathsf{AVal}', \sigma_l \in \mathsf{LStore}, S_1, S_2 \in \mathcal{P}(\mathrm{SState})$, $p1_h, p2_h \in View_{\mathcal{A}}$ such that $\tau_1 \bowtie \tau_2$ is well-defined and $(\tau_1 \bowtie \tau_2)[0] = (h, \_, \_)$,

$$\textbf{if } \left(\begin{array}{c} \tau_1 \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v), S_1 \textbf{ and } \tau_2 \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2 \textbf{ and} \\ h \in [\![p1_h * p2_h * p1_a(v) * p2_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda \end{array}\right)$$
$$\textbf{then } \tau_1 \bowtie \tau_2 \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S$$

where

$$S = \{\ (q1_h * q2_h, p1_a * p2_a, v') \mid (q1_h, p1_a, v') \in S_1 \wedge (q2_h, p2_a, v') \in S_2\ \}$$

and $p1_a = p2_a = \lambda x.\mathtt{Emp}_{\mathcal{A}}$, and I also write $p1_a * p2_a$ to denote $\lambda x.\ p1_a(x) * p2_a(x)$.

Proof. Take $\tau_1, \tau_2 \in \mathsf{Trace}, v \in \mathsf{AVal}', \sigma_l \in \mathsf{LStore}, S_1, S_2 \in \mathcal{P}(\mathrm{SState}), p1_h, p2_h \in View_{\mathcal{A}}, p1_h, p2_h \in View_{\mathcal{A}}$ arbitrary, assume $\tau_1 \bowtie \tau_2$ is well-defined, let it be τ and its first heap h, and assume

$$\tau_1 \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v), S_1 \tag{C.3}$$

$$\tau_2 \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2 \tag{C.4}$$

$$h \in [\![p1_h * p2_h * p1_a(v) * p2_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda \tag{C.5}$$

First consider the case $\tau = (\sigma, h, C)$. From the definition of $\bowtie$, find that $\tau_1 = (\sigma, h, C_1)$ and $\tau_2 = (\sigma, h, C_2)$ where $C = C_1 \;||\; C_2$. From Term, conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), \{(p1_h * p2_h, p1_a * p2_a, v)\}$. Equations C.3, C.4 must also be by an application of Term and thus $S_1 = \{(p1_h, p1_a, v)\}$, $S_2 = \{(p2_h, p2_a, v)\}$, giving the correct relation on $S = \{(p1_h * p2_h, p1_a * p2_a, v)\}$.

Next we examine separately the case the first step in τ is an environment step versus a local step. For environment steps, assume $\tau = \tau_1 \bowtie \tau_2 = (\sigma, h, C)\mathrm{env}(\sigma, h', C)\tau'$ and therefore from the definition of $\bowtie$ find that $\tau_1 = (\sigma, h, C_1)\mathrm{env}(\sigma, h', C_1)\tau_1'$ and $\tau_2 = (\sigma, h, C_2)\mathrm{env}(\sigma, h', C_2)\tau_2'$, $C = C_1 \;||\; C_2$ and $\tau' = \tau_1' \bowtie \tau_2'$. We split on whether $v \in \mathsf{AVal}$ or $v \in \mathsf{AVal} \times \mathsf{AVal}$.

If $v \in \mathsf{AVal}$, from C.3 and C.4 we have

$$\forall v' \in \mathsf{AVal}.\ E(v') \implies (\sigma, h', C_1)\tau_1' \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v'), S_{1v'} \tag{C.6}$$

$$\forall v' \in \mathsf{AVal}.\ F(v') \implies (\sigma, h', C_2)\tau_2' \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v'), S_{2v'} \tag{C.7}$$

where

$$E(v') \triangleq \exists p1_e, p1_e'.\ h \in [\![p1_h * p1_a(v) * p1_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p1_a(v) * p1_e \twoheadrightarrow p1_a(v') * p1_e'$$

$$F(v') \triangleq \exists p2_e, p2_e'.\ h \in [\![p2_h * p2_a(v) * p2_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p2_a(v) * p2_e \twoheadrightarrow p2_a(v') * p2_e'$$

Let

$$G(v') \triangleq \exists p_e, p_e'.\ h \in [\![p1_h * p2_h * p1_a(v) * p2_a(v) * p_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p1_a * p2_a(v) * p_e \twoheadrightarrow p1_a(v') * p2_a(v') * p_e'$$

Take $v' \in \mathsf{AVal}$ arbitrary and assume $G(v')$. See that $G(v') \implies E(v') \wedge F(v')$ by in the first case taking $p1_e = p_e * p2_h * p2_a(v)$ and $p1_e' = p_e * p2_h * p2_a(v')$ and in the second $p2_e = p_e * p1_h * p1_a(v)$ and $p2_e' = p_e * p1_h * p1_a(v')$ - it is clear this yields the first conjunct of $E(v')$ and $F(v')$. As $pi_a(v') = \mathtt{Emp}_{\mathcal{A}} \in View_{\mathcal{A}}$ and $pi_h \in View_{\mathcal{A}}$, Lemma 4.5 gives the second conjunct of $E(v')$ and $F(v')$.
Therefore, from Equations C.6 and C.7, $(\sigma, h', C_1)\tau_1' \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v'), S_{1v'}$ and $(\sigma, h', C_2)\tau_2' \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v'), S_{2v'}$. From the inductive hypothesis, conclude that

$$\forall v' \in \mathsf{AVal}.\ G(v') \implies \exists S_{v'}'.\ (\sigma, h, C)\tau' \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v'), S_{v'}$$

such that

$$\forall v' \in \mathsf{AVal}.\ S_{v'} = \{\ (q1_h * q2_h, p1_a * p2_a, v) \mid (q1_h, p1_a, v) \in S_{1v'} \wedge (q2_h, p2_a, v) \in S_{2v'}\ \} \tag{C.8}$$

By application of Env, $\tau = \tau_1 \bowtie \tau_2 \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S$ where $S = \bigcup_{\forall v' \in X.\ G(v')} S_{v'}$. Therefore, we have that

$$S \subseteq \{\ (q1_h * q2_h, p1_a * p2_a, v) \mid (q1_h, p1_a, v) \in S_1 \wedge (q2_h, p2_a, v) \in S_2\ \} \tag{C.9}$$

In order to check the other direction it is sufficient to check that $E(v') \wedge F(v') \implies G(v')$.

From $E(v') \wedge F(v')$, i.e. find $p1_e, p1'_e, p2_e, p2'_e$ such that

$$h \in [\![p1_h * p1_a(v) * p1_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p1_a(v) * p1_e \twoheadrightarrow p1_a(v') * p1'_e$$

$$h \in [\![p2_h * p2_a(v) * p2_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p2_a(v) * p2_e \twoheadrightarrow p2_a(v') * p2'_e$$

From C.5, we can find some p_e such that

$$h \in [\![p1_h * p1_a(v) * p2_h * p2_a(v) * p_e]\!]_\lambda$$

Therefore deduce that

$$p1_e = p2_h * p2_a(v) * p_e (= p2_h * p_e)$$

$$p2_e = p1_h * p1_a(v) * p_e (= p1_h * p_e)$$

with the second equality from $p1_a(v) = p2_a(v) = \mathtt{Emp}_{\mathcal{A}}$. We need to show that

$$\exists p'_e.\ (h, h') \models_{\lambda,\mathcal{A}} p1_a(v) * p2_a(v) * p_e \twoheadrightarrow p1_a(v') * p2_a(v') * p'_e \tag{C.10}$$

Given $p1_a(v) = p2_a(v) = p1_a(v') = p2_a(v') = \mathtt{Emp}_{\mathcal{A}}$, it suffices to show $\exists p'_e.\ (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e$, given that from $E(v'), F(v')$ and our equations on the sets of worlds

$$(h, h') \models_{\lambda,\mathcal{A}} p2_h * p_e \twoheadrightarrow p1'_e \tag{C.11}$$

$$(h, h') \models_{\lambda,\mathcal{A}} p1_h * p_e \twoheadrightarrow p2'_e \tag{C.12}$$

Take $f \in View_{\mathcal{A}}$ arbitrary such that $h \in [\![p_e * f]\!]_\lambda$. From Equation C.5, $f = p1_h * p2_h$. As $p1_h, p2_h \in View_{\mathcal{A}}$, from Equations C.11, C.12 deduce $h' \in [\![p1'_e * p1_h]\!]_\lambda$ and $h' \in [\![p2'_e * p2_h]\!]_\lambda$. From the definition of reification and world composition, we are free to separate $p1'_e = p'_e * p2_h$ and $p2'_e = p'_e * p1_h$ for some p'_e, precisely because from C.5 we know that $p1_h * p2_h$ is well-defined, and thus they are separable and have no overlap. This p'_e is precisely such which satisfies Equation C.10. From here, we are free to deduce that Equation C.9 is an equality and we have the result.

In the case $v \in \mathsf{AVal} \times \mathsf{AVal}$, it must have been from C.3 and C.4 that

$$\text{if } \exists p1_e, p1'_e.\ h \in [\![p1_h * p1_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p1_e \twoheadrightarrow p1'_e \text{ then } (\sigma, h', \mathbb{C})\tau'_1 \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v), S_1 \text{ else } S_1 = \emptyset \tag{C.13}$$

$$\text{if } \exists p2_e, p2'_e.\ h \in [\![p2_h * p2_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p2_e \twoheadrightarrow p2'_e \text{ then } (\sigma, h', \mathbb{C})\tau'_2 \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2 \text{ else } S_2 = \emptyset \tag{C.14}$$

We use these to prove

$$\text{if } \exists p_e, p'_e.\ h \in [\![p1_h * p2_h * p_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e \text{ then } (\sigma, h', \mathbb{C})\tau \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S_2 \text{ else } S_2 = \emptyset \tag{C.15}$$

In particular, assume $\exists p_e, p'_e.\ h \in [\![p1_h * p2_h * p_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e$. Then as this implies the premises of Equations C.13, C.14, find $(\sigma, h', \mathbb{C})\tau'_1 \models_{\mathbb{S}_1} \sigma_l, (p1_h, p1_a, v), S_1$ and $(\sigma, h', \mathbb{C})\tau'_2 \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2$. From the inductive hypothesis then, $(\sigma, h', C)\tau' \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S$, where

$$S \subseteq \{\ (q1_h * q2_h, p1_a * p2_a, v) \mid (q1_h, p1_a, v) \in S_1 \wedge (q2_h, p2_a, v) \in S_2\ \}$$

Now assume $\neg(\exists p_e, p'_e.\ h \in [\![p1_h * p2_h * p_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e)$. We need to check that this implies $S_1 = \emptyset \vee S_2 = \emptyset$. from Equations C.13, C.14. This amounts to proving

$$\begin{aligned}(\exists p1_e, p1'_e.\ h \in [\![p1_h * p1_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p1_e \twoheadrightarrow p1'_e) \wedge \\ (\exists p2_e, p2'_e.\ h \in [\![p2_h * p2_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p2_e \twoheadrightarrow p2'_e) \implies \\ (\exists p_e, p'_e.\ h \in [\![p1_h * p2_h * p_e]\!]_\lambda \wedge (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e)\end{aligned}$$

Given that $p1_a(v) = p2_a(v) = p1_a(v') = p2_a(v') = \mathtt{Emp}_{\mathcal{A}}$ this is exactly the statement $E(v') \land F(v') \implies G(v')$ which we proved in the previous case.

Consider the case that the first step of τ is a local step, that is, $\tau = (\sigma, h, C)\mathrm{loc}(\sigma, h', C')\tau'$. We know already from the definition of $\bowtie$ that the store may not change, and that as $\tau = \tau_1 \bowtie \tau_2$, either this corresponds to a local step of τ_1 or a local step of τ_2. WLOG we consider the following scenario: $\tau_1 = (\sigma, h, C_1)\mathrm{loc}(\sigma, h', C_1')\tau_1'$, $\tau_2 = (\sigma, h, C_2)\mathrm{env}(\sigma, h', C_2)\tau_2'$ and therefore the following relations hold: $C = C_1 \;||\; C_2$ and $C' = C_1' \;||\; C_2$. Equation C.3 is either from an application of Stutter or LinPt - we do the Stutter case, the LinPt case is completely analagous. Then,

$$(\sigma, h', C_1')\tau_1' \models_{\mathbb{S}_1} \sigma_l, (p1_h', p1_a, v), S_1 \tag{C.16}$$
$$(h, h') \models_{\lambda,\mathcal{A}} p1_h * p1_a(v) \twoheadrightarrow p1_h' * p1_a(v) \tag{C.17}$$
$$\mathbb{C}_1' = \checkmark \implies v \in \mathsf{AVal} \times \mathsf{AVal} \land p1_h' = \mathcal{W}[\![Q_1(v)]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma} \tag{C.18}$$

and from hypothesis C.4 in the case $v \in \mathsf{AVal}$ Env gives,

$$\begin{aligned} &E(v') \triangleq \exists p_e, p_e'.\ h \in [\![p2_h * p2_a(v) * p_e]\!]_\lambda \land (h, h') \models_{\lambda,\mathcal{A}} p2_a(v) * p_e \twoheadrightarrow p2_a(v') * p_e' \\ &\forall v' \in \mathsf{AVal}.\ E(v') \implies (\sigma, h', C_2)\tau_2' \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v'), S_{v'} \\ &S_1 = \bigcup_{v' \in \mathsf{AVal}.\ E(v')} S_{v'} \end{aligned} \tag{C.19}$$

We aim to prove:

$$(\sigma, h', C_1' \;||\; C_2)\tau' \models_{\mathbb{S}} \sigma_l, (p1_h' * p2_h, p1_a * p2_a, v), S \tag{C.20}$$
$$S = \{\ (q1_h * q2_h, p1_a * p2_a, v) \mid (q1_h, p1_a, v) \in S_1 \land (q2_h, p2_a, v) \in S_2\ \} \tag{C.21}$$
$$(h, h') \models_{\lambda,\mathcal{A}} p1_h * p2_h * p1_a(v) * p2_a(v) \twoheadrightarrow p1_h' * p2_h * p1_a(v) * p2_a(v) \tag{C.22}$$
$$C_1' \;||\; C_2 = \checkmark \implies v \in \mathsf{AVal} \times \mathsf{AVal} \land p1_h' * p2_h = \mathcal{W}[\![Q_1 * Q_2]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma} \tag{C.23}$$

Clearly $C_1' \;||\; C_2 \neq \checkmark$ so premise C.23 is immediate. As $p2_a(v) = \mathtt{Emp}_{\mathcal{A}} \in View_{\mathcal{A}}$ and $p1_h \in View_{\mathcal{A}}$, we can directly conclude that Equation C.17 implies C.22. It remains to show Equations C.20 and C.21. Given C.5, we know that $\exists p_e.\ h \in [\![p2_h * p2_a(v) * p_e]\!]$ and furthermore that $p_e = p1_h * p1_a(v)$. Equation C.17 implies that $h' \in [\![p1_h' * p2_h * p1_a(v) * p2_a(v)]\!]_\lambda$ (as $p2_h, p2_a(v) \in View_{\mathcal{A}}$). Thus $E(v)$ holds and by Equation C.19 find S_v such that $(\sigma, h', C_2)\tau_2' \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_v$. From this, Equation C.16 and the inductive hypothesis deduce Equations C.20 and C.22 hold - the equality of C.22 is given from the necessity of matching the AVal's.

In the case $v \in \mathsf{AVal} \times \mathsf{AVal}$, Equation C.4 gives

$$\text{if } \exists p_e, p_e'.\ h \in [\![p2_h * p_e]\!]_\lambda \land (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p_e' \text{ then } (\sigma, h', C_2)\tau_2' \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2 \text{ else } S_2 = \emptyset \tag{C.24}$$

and again we aim to prove premises C.20, C.21, C.22 and C.23. As before, C.23 is immediate and Equation C.17 implies C.22. Equation C.5 allows us to take $p_e = p1_h * p1_a(v) * p2_a(v)$ and C.23 implies that for $p_e' = p1_h * p1_a(v) * p2_a(v)$ then $h \in [\![p2_h * p_e]\!]_\lambda \land (h, h') \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p_e'$ holds and thus by Equation C.24 $(\sigma, h', C_2)\tau_2' \models_{\mathbb{S}_2} \sigma_l, (p2_h, p2_a, v), S_2$. Find by the inductive hypothesis that Equations C.20 and C.21 hold.

Finally, Equations C.20, C.22 and C.23 give us the necessary premises to apply Stutter and conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p1_h * p2_h, p1_a * p2_a, v), S$ with S as in Equation C.21, for any $v \in \mathsf{AVal}'$. □

Lemma (6.9 Trace safety is closed under appending safe local steps)**.** Let $\mathbb{S} \in \mathsf{Spec}$, $f : \mathrm{SState} \to View_{\mathcal{A}}$, $g : \mathrm{SState} \to \mathsf{AVal}$ and $\tau(\sigma_1, h_1, \mathbb{C}_1)$ loc $(\sigma_2, h_2, \mathbb{C}_2) \in \mathsf{Trace}$. The following hold:

1. **if** $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ **and**
$\forall (p_h', p_a, v') \in S.\ (h_1, h_2) \models_{\lambda,\mathcal{A}} p_h' * p_a(v') \twoheadrightarrow f(p_h', p_a, v') * p_a(v')$ **and**
(**if** $\mathbb{C}_2 = \checkmark$ **then** $f(p_h', p_a, v') = \mathcal{W}[\![Q_h(v')]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma_2}$ **and** $v' \in \mathsf{AVal} \times \mathsf{AVal}$)
then $\tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{\ (f(p_h', p_a, v'), p_a, v') \mid (p_h', p_a, v') \in S\ \}$

2. **if** $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ **and**
$\forall (p_h', p_a, v') \in S.\ (h_1, h_2) \models_{\lambda,\mathcal{A}} p_h' * p_a(v') \twoheadrightarrow f(p_h', p_a, v') * \mathcal{W}[\![Q_a(v, g(p_h', p_a, v'))]\!]_{\mathcal{A}}^{\sigma_l}$ **and**
(**if** $\mathbb{C}_2 = \checkmark$ **then** $f(p_h', p_a, v') = \mathcal{W}[\![Q_h(v')]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma_2}$)
then $\tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{\ (f(p_h', p_a, v'), p_a, g(p_h', p_a, v')) \mid (p_h', p_a, v') \in S\ \}$

Proof. Many parts of this proof echo closely the proof of Lemma 6.2. When proceeding by induction on the structure of the trace safety judgement I do explicitly here only the Term, Stutter and Env cases.
Take $\mathbb{S}, \tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \in \mathsf{Trace}$, $f : \mathsf{SState} \to \mathit{View}_{\mathcal{A}}$ arbitrary and assume that $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ and that $\forall (p'_h, p_a, v') \in S.\ (h_1, h_2) \models_{\lambda,\mathcal{A}} p'_h * p_a(v') \twoheadrightarrow f(p'_h, p_a, v') * p_a(v') \wedge \mathbb{C}_2 = \checkmark \implies f(p'_h, p_a, v') = \mathcal{W}[\![Q_h(v')]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma_2} \wedge v' \in \mathsf{AVal} \times \mathsf{AVal}$. We aim to show $\tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S\ \}$ by induction on the structure of the trace safety judgement.

- **Case: Term**
  Assume that $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ by an application of Term, i.e. we actually have $(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{\ (p_h, p_a, v)\ \}$. From the second assumption, if we let $f(p_h, p_a, v) = p'_h$, find by Term that $(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), \{\ (p'_h, p_a, v)\ \}$ then $(h_1, h_2) \models_{\lambda,\mathcal{A}} p_h * p_a(v') \twoheadrightarrow p'_h * p_a(v') \wedge \mathbb{C}_2 = \checkmark \implies p'_h = \mathcal{W}[\![Q_h(v')]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma_2} \wedge v' \in \mathsf{AVal} \times \mathsf{AVal}$ and thus by Stutter that $(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(p'_h, p_a, v)\}$.

- **Case: Stutter**
  This is an immediate application of the inductive hypothesis. Assume that $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ by an application of Stutter, that is, $(\sigma_0, h_0, \mathbb{C}_0)\mathrm{loc}\tau'(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. Therefore, there is some p'_h such that $\tau'(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S$. An application of the inductive hypothesis yields

  $$\tau'(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), \{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S\ \}$$

  and then reapplying Stutter concludes

  $$\tau(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S\ \}$$

- **Case: Env**
  This is also an immediate application of the inductive hypothesis. Assume that $\tau(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ by an application of Env, that is, $(\sigma_0, h_0, \mathbb{C}_0)\mathrm{env}\tau'(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. The premises of Env give us that $v \in \mathsf{AVal}$,

  $$\forall v' \in X.\ E(v') \implies \tau'(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S_{v'} \tag{C.25}$$

  and the necessary relation between S and the $S_{v'}$s. We need to verify that

  $$\forall v' \in X.\ E(v') \implies \tau'(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), \{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S_{v'}\ \}$$

  Take $v' \in X$ arbitrary and assume $E(v')$. From the premise, find $S_{v'}$ such that $\tau'(\sigma_1, h_1, \mathbb{C}_1) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S_{v'}$. Apply the inductive hypothesis to find that $\tau'(\sigma_1, h_1, \mathbb{C}_1)\mathrm{loc}(\sigma_2, h_2, \mathbb{C}_2) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), \{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S_{v'}\ \}$ and we are done. Reapplying the Env rule again and observing that our S is the union of each $\{\ (f(p'_h, p_a, v'), p_a, v') \mid (p'_h, p_a, v') \in S_{v'}\ \}$, we get our result.

The second conjunct proceeds identically to the first. □

Lemma (6.10 Safe terminated threads satisfy postcondition)**.** Let $\tau(\sigma_n, h_n, \mathbb{C}_n) \in \mathsf{Trace}$, $\mathbb{C} \in \mathsf{Cmd}$, $\mathbb{S} \in \mathsf{Spec}$, φ, $\sigma_l \in \mathsf{LStore}$, $v \in X$ and $S \in \mathcal{P}(\mathsf{SState})$. The following holds:

$$\textbf{if } \tau(\sigma_n, h_n, \mathbb{C}_n) \in [\![\mathbb{C}]\!]_{\varphi} \textbf{ and } \mathbb{C}_n = \checkmark \textbf{ and } \tau(\sigma_n, h_n, \mathbb{C}_n) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$$
$$\textbf{then } \forall (p'_h, p_a, v') \in S.\ p'_h = \mathcal{W}[\![Q_h(v')]\!]_{\mathcal{A}}^{\sigma_n \circ \sigma_l} \textbf{ and } v' \in \mathsf{AVal} \times \mathsf{AVal}$$

Proof. Take $\tau, \mathbb{C}, \mathbb{S}, \varphi, \sigma_l, v$ arbitrary and proceed by induction on the length of τ. As traces of the form $(\sigma, h, \checkmark) \notin [\![\mathbb{C}]\!]_{\varphi}$, we begin with the base case of τ only having one local step: $\tau \in [\![\mathbb{C}]\!]_{\varphi}$ and begin with $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark)$ Assume $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. Clearly this must be an application of LinPt followed by Term if $v \in \mathsf{AVal}$ and Stutter then Term if $v \in \mathsf{AVal} \times \mathsf{AVal}$, thus S is a singleton state $(q_h, p_a, \langle v, v' \rangle)$ such that $q_h = \mathcal{W}[\![Q(h)(v, v')]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$. If $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark)\tau'$ where τ' consists only of environment steps, the assumption must be applications of LinPt followed by a sequence of Env' followed by Term, by definition of the trace safety judgemenet, and Env' does not modify S.
In the inductive case we have $\tau = (\sigma_0, h_0, \mathbb{C}_0)\mathrm{env}(\sigma_0, h_0, \mathbb{C}_0)\tau'$ and the result holds for τ' then examination

of the premises of Env if $v \in \mathsf{AVal}$ (or Env' if $v \in \mathsf{AVal} \times \mathsf{AVal}$) gives the correct result.
So let's inductively consider $(\sigma_0, h_0, \mathbb{C}_0)\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau'$ where the result holds for τ'. From the $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ assumption, this initial step must be safe due to an application of either Stutter or LinPt. But in fact these steps cannot modify any S, so the inductive hypothesis immediately gives us the result. □

Lemma C.1. For $h_0, h_1 \in \mathsf{Heap}, p_h \in View_{\mathcal{A}}, v \in X$,

$$\textbf{if } (h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p'_a(v) \twoheadrightarrow p'_h * p'_a(v) \textbf{ then } (h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * p_a(v)$$

Proof. Take $h_0, h_1 \in \mathsf{Heap}, p_h \in View_{\mathcal{A}}, v \in X, f \in View_{\mathcal{A}}$ and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_{\lambda+1}$. Find $w_h \in p_h, w_a \in p_a(v), w_f \in f$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_{\lambda+1}$. See that for $w_h = (\_, \rho_h, \_, \_)$, $w_a = (\_, \rho_a, \_, \_)$, $w_f = (\_, \rho_f, \_, \_)$, from the definition of world composition it must be that $\rho_h = \rho_a = \rho_f$, and let this be ρ going forward. If $\mathrm{closed}_\lambda^{\lambda+1}(\rho) = \{r, r_1, ..., r_n\}$ (note that the definition of $p_a(v)$ requires $r \in \mathrm{closed}_\lambda^{\lambda+1}(\rho)$), then let $\mathbf{t}_i \in \mathsf{RId}, a_i \in \mathsf{AVal}$ be such that $\rho(r_i) = (\mathbf{t}_i, \lambda, a_i)$ (and $p_a(v)$ dictates that $\rho(r) = (\mathbf{t}, \lambda, v)$). The definition of reification dictates that $\exists w_r \in \mathcal{I}_{\mathbf{t}}[\![r, \lambda, v]\!], w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \bullet w_r \bullet w_1 \bullet ... \bullet w_n \rfloor_\lambda$. From the definition of $p'_a(v)$, find that $w_a \bullet w_r \in p'_a(v)$, and conclude that $h_0 \in [\![p_h * p'_a(v) * f * \circledast_{1 \le i \le n} \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]]\!]_\lambda$. As region interpretations are required to be stable, we can now apply our original hypothesis to find that

$$h_1 \in [\![p'_h * p'_a(v) * f * \circledast_{1 \le i \le n} \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]]\!]_\lambda$$

Returning back to the definition of reification finds $\exists\ \overline{w}'_h \in p'_h, \overline{w}'_a \in p'_a(v), \overline{w}_f \in f, \overline{w}_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]$ such that $h_1 \in \lfloor \overline{w}'_h \bullet \overline{w}'_a \bullet \overline{w}_f \bullet \overline{w}_1 \bullet ... \bullet \overline{w}_n \rfloor_\lambda$, and from the definition of $p'_a(v)$ find some $\overline{w}_a \in \mathcal{W}[\![P_a(v) * \lceil G \rceil_r * v \in X]\!]^{\sigma_l}_{\mathcal{A}}$ and $\overline{w}_r \in \mathcal{I}_{\mathbf{t}}[\![r, \lambda, v]\!]$ such that $\overline{w}_a \bullet \overline{w}_r = \overline{w}'_a$. Conclude that

$$h_1 \in \lfloor \overline{w}'_h \bullet \overline{w}_a \bullet \overline{w}_r \bullet \overline{w}_f \bullet \overline{w}_1 \bullet ... \bullet \overline{w}_n \rfloor_\lambda$$

Similarly to above, the shared regions of each world in the reification are represented by the same ρ'. The definition of frame-preserving updates is designed to only permit modifications to ρ in such a way which is consistent with the logical ghost state held (in general, this can be seen by taking the frame f to be the closure of each region with a lower level, $\mathcal{W}[\![\mathbf{t}^\lambda_r(x)]\!]^\emptyset_{\mathcal{A}}$ under the rely relation and reasoning about the changes to ρ permitted). By applying the stability requirements of region interpretations, we find that $\overline{w}_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a'_i]\!]$, where $\rho'(r_i) = (\mathbf{t}_i, \lambda, a'_i)$. As p'_a does not specify a region r, we can use the upwards-closed property to augment $\rho'(r)$ with $r \mapsto [\mathbf{t}, \lambda, v]$. Finally, applying reification to this to go back up a level now that our region interpretations are consistent with the logical representation in the worlds finds that

$$h_1 \in [\![p'_h * p_a(v) * f]\!]_\lambda$$

□

Lemma C.2. For $h_0, h_1 \in \mathsf{Heap}, p_h \in View_{\mathcal{A}}, \sigma_l \in \mathsf{LStore}, v \in X$,

$$\textbf{if } (h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p'_a(v) \twoheadrightarrow p'_h * q'_a(x, y) \textbf{ then } (h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * q_a(x, y)$$

where

$$q_a = \lambda x, y.\ \mathcal{W}[\![\exists z.\ Q_a(x, y, z) \star \mathbf{t}^\lambda_r(z) \star R(x, z)]\!]^{\sigma_l}_{\mathcal{A}}$$
$$q'_a = \lambda x, y.\ \mathcal{W}[\![\exists z.\ Q_a(x, y, z) \star I(\mathbf{t}^\lambda_r(z)) \star R(x, z)]\!]^{\sigma_l}_{\mathcal{A}}$$

Proof. Proceeds identically to the proof of Lemma C.1. □

Lemma C.3. For $h_0, h_1 \in Heap, p_h \in View_{\mathcal{A}}, v \in X, v' \in \mathsf{AVal}$,

$$\textbf{if } (\exists p_e, p'_e.\ h_0 \in [\![p_h * p_a(v) * p_e]\!]_{\lambda+1} \textbf{ and } (h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p'_e)$$
$$\textbf{then } (\exists p_f, p'_f.\ h_0 \in [\![p_h * p'_a(v) * p_f]\!]_\lambda \textbf{ and } (h_0, h_1) \models_{\lambda,\mathcal{A}} p'_a(v) * p_f \twoheadrightarrow p'_a(v') * p'_f)$$

Proof. Take $h_0, h_1 \in \mathsf{Heap}, p_h \in View_{\mathcal{A}}, v \in X, v' \in \mathsf{AVal}$ arbitrary and find some p_e, p'_e such that

$$h_0 \in [\![p_h * p_a(v) * p_e]\!]_{\lambda+1} \tag{C.26}$$
$$(h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p'_e \tag{C.27}$$

We aim to prove the following:

$$h_0 \in [\![p_h * p'_a(v) * p_f]\!]_\lambda \tag{C.28}$$
$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p'_a(v) * p_f \mathrel{-\!\!*} p'_a(v') * p'_f \tag{C.29}$$

for the following definition of p_f, p'_f:

$$p_f = \left\{ w_e \bullet w_1 \bullet ... \bullet w_n \;\middle|\; \begin{array}{l} w_e = (\_, \rho_e, \_, \_) \in p_e, \mathrm{closed}^{\lambda+1}_\lambda(\rho_e) = \{r_1, ..., r_n\} \\ \rho_e(r_i) = (\mathbf{t}_i, \lambda, a_i) \wedge w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!] \end{array} \right\}$$
$$p'_f = \left\{ w_e \bullet w_1 \bullet ... \bullet w_n \;\middle|\; \begin{array}{l} w_e = (\_, \rho_e, \_, \_) \in p'_e, \mathrm{closed}^{\lambda+1}_\lambda(\rho_e) = \{r_1, ..., r_n\} \\ \rho_e(r_i) = (\mathbf{t}_i, \lambda, a_i) \wedge w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!] \end{array} \right\}$$

From Equation C.26, find $w_h \in p_h, w_a \in p_a(v), w_e \in p_e$ such that $h_0 \in \lfloor \overline{w}_h \bullet \overline{w}_a \bullet w_e \rfloor_{\lambda+1}$ and they all share a common ρ. Let $\mathrm{closed}^{\lambda+1}_\lambda(\rho) = \{r, r_1, ..., r_n\}$ and $\rho(r_i) = (\mathbf{t}_i, \lambda, a_i), \rho(r) = (\mathbf{t}, \lambda, v)$. From the definition of reification, $h_0 \in \lfloor w_h \bullet w_a \bullet w_r \bullet w_e \bullet w_1 \bullet ... \bullet w_n \rfloor_\lambda$ for some $w_r \in \mathcal{I}_{\mathbf{t}}[\![r, \lambda, v]\!], w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]$. $w_e \bullet w_1 \bullet ... \bullet w_n \in f$ (let this be w_f), so $h_0 \in \lfloor w_h \bullet (w_a \bullet w_r) \bullet w_f \rfloor_\lambda$ and conclude $h_0 \in [\![p_h * p'_a(v) * p_f]\!]_\lambda$ - exactly Equation C.28 and as $w_a \bullet w_r \in p'_a(v)$.
Let's take $g \in View_\mathcal{A}$ and assume $h_0 \in [\![p'_a(v) * p_f * g]\!]_\lambda$, that is, find some $w'_a \in p'_a(v), w'_f \in p_f, w'_g \in g$ such that $h_0 \in \lfloor w'_a \bullet w'_f \bullet w'_g \rfloor_\lambda$. From the definitions of p_a and p'_a, it is clear we should be able to find some $w''_a = (h_a, \rho''_a, \gamma_a, \chi_a) \in \mathcal{W}[\![P_a(v) \star \lceil G \rceil_r \star v \in X]\!]^{\sigma_l}_\mathcal{A}$, $w'_r \in \mathcal{I}_\mathbf{t}[\![r, \lambda, v]\!]$ such that $w'_a = w''_a \bullet w'_r$. The upwards-closed nature of worlds implies $\exists w_a = (h_a, \rho_a, \gamma_a, \chi_a) \in p_a$ such that $\rho_a = \rho''_a[r \mapsto (\mathbf{t}, \lambda, v)]$, and similarly for w_i, w_g, w_r, such that

$$h_0 \in \lfloor w_a \bullet w_e \bullet w_r \bullet w_1 \bullet ... \bullet w_n \bullet w_g \rfloor_\lambda$$

Reification tells us that

$$h_0 \in \lfloor w_a \bullet w_e \bullet w_g \rfloor_{\lambda+1}$$

and then application of the hypothesis that

$$h_1 \in [\![p_a(v') * p'_e * g]\!]_{\lambda+1}$$

It remains to show $h_1 \in [\![p'_a(v') * p'_f * g]\!]_\lambda$. Find $\overline{w}_a \in p_a(v'), \overline{w}_e \in p'_e, \overline{w}_g \in g, \overline{w}_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!], \overline{w}_r \in \mathcal{I}_\mathbf{t}[\![r, \lambda, v']\!]$ such that $h_1 \in \lfloor \overline{w}_a \bullet \overline{w}_e \bullet \overline{w}_g \bullet w_r \bullet \overline{w}_1 \bullet ... \bullet \overline{w}_n \rfloor_\lambda$ and that $\overline{w}_a = (\_, \overline{rho}, \_, \_) \bullet \overline{w}_r \in p'_a(v')$, where $r \notin dom(\overline{\rho})$. Furthermore, $w_e \bullet w_1 \bullet ... \bullet w_n \in p'_f$. Conclude $h_1 \in [\![p'_a(v') * p'_f * g]\!]_{\lambda+1}$. □

Lemma C.4. For $h_0, h_1 \in \mathsf{Heap}, p_h \in View_\mathcal{A}$,

$$\textbf{if } (\exists p_e, p'_e.\ h_0 \in [\![p_h * p_e]\!]_{\lambda+1} \textbf{ and } (h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_e \mathrel{-\!\!*} p'_e)$$
$$\textbf{then } (\exists p_f, p'_f.\ h_0 \in [\![p_h * p_f]\!]_\lambda \textbf{ and } (h_0, h_1) \models_{\lambda,\mathcal{A}} p_f \mathrel{-\!\!*} p'_f)$$

Proof. Take $h_0, h_1 \in \mathsf{Heap}, p_h \in View_\mathcal{A}$ arbitrary and find some p_e, p'_e such that

$$h_0 \in [\![p_h * p_e]\!]_{\lambda+1} \tag{C.30}$$
$$(h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_e \mathrel{-\!\!*} p'_e \tag{C.31}$$

Given the definition of p_f, p'_f below, we aim to prove Equations C.32 and C.33.

$$p_f = \left\{ w_e \bullet w_1 \bullet ... \bullet w_n \;\middle|\; \begin{array}{l} w_e = (\_, \rho_e, \_, \_) \in p_e, \mathrm{closed}^{\lambda+1}_\lambda(\rho_e) = \{r_1, ..., r_n\} \\ \rho_e(r_i) = (\mathbf{t}_i, \lambda, a_i) \wedge w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!] \end{array} \right\}$$
$$p'_f = \left\{ w_e \bullet w_1 \bullet ... \bullet w_n \;\middle|\; \begin{array}{l} w_e = (\_, \rho_e, \_, \_) \in p'_e, \mathrm{closed}^{\lambda+1}_\lambda(\rho_e) = \{r_1, ..., r_n\} \\ \rho_e(r_i) = (\mathbf{t}_i, \lambda, a_i) \wedge w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!] \end{array} \right\}$$
$$h_0 \in [\![p_h * p_f]\!]_\lambda \tag{C.32}$$
$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_f \mathrel{-\!\!*} p'_f \tag{C.33}$$

Find $w_h \in p_h, w_e \in p_e$ such that $h_0 \in \lfloor w_h \bullet w_e \rfloor_{\lambda+1}$ from Equation C.30, and for $w_h = (\_, \rho_h, \_, \_)$, $w_e = (\_, \rho_e, \_, \_)$, see that $\rho_h = \rho_e$. Let $\mathrm{closed}^{\lambda+1}_\lambda(\rho) = \{r_1, ..., r_n\}$ and $\rho(r_i) = (\mathbf{t}_i, \lambda, a_i)$. From the definition of reification it is clear that there are some $w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]$ such that $h_0 \in \lfloor w_h \bullet w_e \bullet w_1 \bullet ... \bullet w_n \rfloor_\lambda$, and that $w_e \bullet w_1 \bullet ... \bullet w_n \in p_f$ so $h_0 \in [\![p_h * p_f]\!]_\lambda$ - exactly Equation C.32.
So let's take $g \in View_\mathcal{A}$ and assume $h_0 \in [\![p_f * g]\!]_\lambda$, i.e. find some $w_f = (\_, \rho_f, \_, \_) \in p_f, w_g =$

$(\_, \rho_g, \_, \_) \in g$ such that $h_0 \in \lfloor w_f \bullet w_g \rfloor_\lambda$ and $\rho_f = \rho_g$ - let this be ρ. Let $\text{closed}_\lambda^{\lambda+1}(\rho) = \{r_1, ..., r_n\}$ and $\rho(r_i) = (\mathbf{t}_i, \lambda, a_i)$. The construction of p_f is such that $\exists w_e = (\_, \rho_e, \_, \_) \in p_e, w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]$ such that $w_f = w_e \bullet w_1 \bullet ... \bullet w_n$, $\rho_e = \rho$.
We have $h_0 \in \lfloor w_e \bullet w_1 \bullet ... \bullet w_n \bullet w_g \rfloor_\lambda$, and from the definition of ρ therefore $h_0 \in \lfloor w_e \bullet w_g \rfloor_{\lambda+1}$. Conclude $h_0 \in [\![p_e * g]\!]_{\lambda+1}$, from assumption C.31 that $h_1 \in [\![p'_e * g]\!]_{\lambda+1}$ and find some $w'_e = (\_, \rho'_e, \_, \_) \in p'_e, w'_g = (\_, \rho'_g, \_, \_) \in g$ such that $h_1 \in \lfloor w'_e \bullet w'_g \rfloor_{\lambda+1}$ and $\rho'_e = \rho'_g$ - let this be ρ'. For $\text{closed}_\lambda^{\lambda+1}(\rho') = \{r_1, ..., r_n\}$ and $\rho'(r_i) = (\mathbf{t}_i, \lambda, a'_i)$, from the definition of reification it is immediate that $\exists w'_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a'_i]\!]$ such that $h_1 \in \lfloor w'_e \bullet w'_g \bullet w'_1 \bullet ... \bullet w'_n \rfloor_\lambda$. See that $w'_e \bullet w'_1 \bullet ... \bullet w'_n \in p_f$ to conclude that $h_1 \in [\![p'_f * g]\!]_\lambda$. □

Lemma C.5. For $\tau \in \mathsf{Trace}, \sigma_l \in \mathsf{LStore}, v \in X, p_h \in View_{\mathcal{A}}$,

$$\textbf{if } \tau \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S' \textbf{ then } \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$$

where $S \subseteq S'$.

Proof. Once again, we prove this by induction on the trace safety judgement. Take $\tau \in \mathsf{Trace}, \sigma_l \in \mathsf{LStore}, v \in X, p_h \in View_{\mathcal{A}}$ arbitrary and assume $\tau \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$.

- **Case: Term** Immediate from the definition of Term.
- **Case: Stutter** Then $(\sigma_0, h_0, \mathbb{C}_0)\text{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$ and from the premises we have

$$(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S' \tag{C.34}$$
$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p'_a(v) \twoheadrightarrow p'_h * p'_a(v) \tag{C.35}$$
$$\mathbb{C}_1 = \checkmark \implies v \in \mathsf{AVal} \times \mathsf{AVal} \wedge p'_h = \mathcal{W}[\![Q_h(v)]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma_1} \tag{C.36}$$

  Right away the inductive hypothesis tells us that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S$ for $S \subseteq S'$. Given the Hoare postcondition of $\mathbb{S}$ and $\mathbb{S}'$ coincide, Equation C.36 will be immediately applicable. It remains to show that

$$(h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * p_a(v) \tag{C.37}$$

  This is given by application of Lemma C.1 to Equation C.35.
  So we have checked the necessary premises for Stutter, and may conclude that

$$(\sigma_0, h_0, \mathbb{C}_0)\text{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$$

- **Case: LinPt** Then $(\sigma_0, h_0, \mathbb{C}_0)\text{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$. Let

$$q_a = \lambda x, y.\ \mathcal{W}[\![\exists z.\ Q_a(x, y, z) \star \mathbf{t}_r^\lambda(z) \star R(x, z)]\!]_{\mathcal{A}}^{\sigma_l}$$
$$q'_a = \lambda x, y.\ \mathcal{W}[\![\exists z.\ Q_a(x, y, z) \star I(\mathbf{t}_r^\lambda(z)) \star R(x, z)]\!]_{\mathcal{A}}^{\sigma_l}$$

  and from the premises we have

$$(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, \langle v, v' \rangle), S' \tag{C.38}$$
$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p'_a(v) \twoheadrightarrow p'_h * q'_a(v, v') \tag{C.39}$$
$$\mathbb{C}_1 = \checkmark \implies p'_h = \mathcal{W}[\![Q_h(v, v')]\!]_{\mathcal{A}}^{\sigma_l \circ \sigma_1} \tag{C.40}$$

  The inductive hypothesis and Equation C.38 finds that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, \langle v, v' \rangle), S$, where $S \subseteq S'$. The two specifications share a postcondition, so Equation C.40 is immediately usable. It remains to check

$$(h_0, h_1) \models_{\lambda+1,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * q_a(v, v') \tag{C.41}$$

  This is exactly given by Lemma C.2 and Equation C.39. Conclude by application of LinPt that $\tau \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S$.

- **Case: Env** Then $(\sigma_0, h_0, \mathbb{C}_0)\text{env}(\sigma_0, h_1, \mathbb{C}_0)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$ and from the premises we have

$$v \in \mathsf{AVal}$$
$$E(v') \triangleq \exists p_e, p'_e.\ h_0 \in [\![p_h * p'_a(v) * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p'_a(v) * p_e \twoheadrightarrow p'_a(v') * p'_e$$
$$\forall v' \in X.\ E(v') \implies (\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v'), S' \tag{C.42}$$

Let

$$F(v') \triangleq \exists \overline{p}_e, \overline{p}'_e.\ h_0 \in [\![p_h * p_a(v) * p_e]\!]_{\lambda+1} \wedge (h_0, h_1) \models_{\lambda+1,\mathcal{A}} p'_a(v) * p_e \mathrel{-\!\!*\!\!>} p_a(v') * p'_e$$

Take $v' \in X$ arbitrary and assume $F(v')$. Lemma C.3 tells us $E(v')$ holds, so from Equation C.42 find that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v'), S'$. The inductive hypothesis says $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v'), S$, where $S \subseteq S'$ and so finally application of Env yields $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \overline{S}$, where $\overline{S} \subset S \subseteq S'$.

- **Case: Env'** Then $(\sigma_0, h_0, \mathbb{C}_0)\mathrm{env}(\sigma_0, h_1, \mathbb{C}_0)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$ and from the premises we have

$$\begin{aligned} &\text{if } \exists p_e, p'_e.\ h_0 \in [\![p_h * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_e \mathrel{-\!\!*\!\!>} p'_e \\ &\qquad \text{then } (\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S' \text{ else } S' = \emptyset && \text{(C.43)} \\ &v \in \mathsf{AVal} \times \mathsf{AVal} && \text{(C.44)} \end{aligned}$$

Assume

$$\exists \overline{p}_e, \overline{p}'_e.\ h_0 \in [\![p_h * p_e]\!]_{\lambda+1} \wedge (h_0, h_1) \models_{\lambda+1,\mathcal{A}} \overline{p}_e \mathrel{-\!\!*\!\!>} \overline{p}'_e \tag{C.45}$$

Then Equation C.45 and Lemma C.4 immediately gives $\exists p_e, p'_e.\ h_0 \in [\![p_h * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_e \mathrel{-\!\!*\!\!>} p'_e$, so from C.43, we find that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$. Immediately from the inductive hyopthesis we find $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ for $S \subseteq S'$, so $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. If Equation C.45 does not hold, then $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \emptyset$ holds and clearly $\emptyset \subseteq S'$, both regardless of the truth of the condition of C.43.

- **Case: Env$\lightning$** Immediate from the definition of Env$\lightning$.

□

Appendix D

Soundness

D.1 Primitives

I do the ASSIGN case in more detail, and then provide only the key steps going forward.

Theorem D.1 (Soundness of ASSIGN)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of ASSIGN}$$
$$\textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}.$$

Proof. Assume that $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by an application of Assign, i.e. $\mathbb{C} = \mathtt{x} := \mathtt{E}$ and $\mathbb{S} = \forall \mathtt{x} \in X.\ \langle\ \mathtt{x} = v \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\langle\ \mathtt{x} = \mathtt{E}[v/\mathtt{x}] \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$. Take $\varphi \in \mathsf{FSpec}$ and assume that $\models \varphi : \Phi$. We aim to show that $[\![\mathbb{C}]\!]_\varphi \subseteq [\![\mathbb{S}]\!]$. For $(\sigma, h, \mathbb{C}) \in [\![\mathbb{C}]\!]$, we find that $\forall \sigma_l, v$, as we can always apply Term, $(\sigma, h, \mathbb{C})$ is safe and so $(\sigma, h, \mathbb{C}) \in [\![\mathbb{S}]\!]$. From Lemma 6.1, we know that for traces of more than just a single configuration, it is sufficient to check the result for traces beginning with a local step. So let's take an arbitrary $\tau \in [\![\mathbb{C}]\!]_\varphi$ (so $\tau = (\sigma_0, h_0, \mathbb{C})\tau'$), with the stipulation that that first step is a local step. We first aim to show that $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark) \in [\![\mathbb{S}]\!]$, and then as there are no further local steps possible, any $\tau \in [\![\mathbb{C}]\!]_\varphi$ beginning with a local step in fact consists solely of this step followed by environment steps. From Lemma 6.2 we find that it is sufficient to check only the $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark) \in [\![\mathbb{S}]\!]$ case and all other τ follows.
To show that $\tau \in [\![\mathbb{S}]\!]$, so take $\sigma_l \in \mathsf{LStore}, v \in X$ arbitrary and assume that $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_\mathcal{A}]\!]_\lambda$, where p_h, p_a are as in the definition of $[\![\mathbb{S}]\!]$. Upon examination of the operational semantics, it's clear that the only possibility for a local step here is $(\sigma_0, h_0, \mathbb{C}) \xrightarrow{\mathrm{loc}}_\varphi (\sigma_1, h_1, \checkmark)$, where $h_0 = h_1$ and $\sigma_1 = \sigma_0[\mathtt{x} \mapsto [\![E]\!]_{\sigma_0}]$.An application of LinPt and then Term would suffice to prove that $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \mathbb{C}) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), \{(q_h, p_a, \langle v, v\rangle)\}$, where $q_h = \mathcal{W}[\![Q_h(v, v)]\!]_\mathcal{A}^{\sigma_2 \circ \sigma_l}$, which then allows us to conclude that $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark) \in [\![\mathbb{S}]\!]$.
From Term, we can trivially obtain that $(\sigma_1, h_1, \checkmark) \models_\mathbb{S} \sigma_l, (q_h, p_a, \langle v, v\rangle), \{(q_h, p_a, \langle v, v\rangle)\}$. Also $v \in \mathsf{AVal}$ and $\checkmark = \checkmark \implies q_h = \mathcal{W}[\![Q_h(v, v)]\!]_\mathcal{A}^{\sigma_2 \circ \sigma_l}$ hold by assumption, so it remains to check

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![Q_a(v, v)]\!]_\mathcal{A}^{\sigma_l} \tag{D.1}$$

Take $f \in View_\mathcal{A}$ arbitrary and assume that $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$. In this particular case, $p_h = \mathcal{W}[\![\mathtt{x} = v]\!]_\mathcal{A}^{\sigma_0 \circ \sigma_l}$ and $p_a(v) = \mathtt{Emp}_\mathcal{A}$. h_0 in this particular set implies nonemptiness, which further implies nonemptiness of p_h. Therefore conclude $\sigma_0(\mathtt{x}) = \sigma_l(v)$ and $p_h = \mathtt{Emp}_\mathcal{A}$ also. So we actually have here that $h_0 \in [\![f]\!]_\lambda$. As $h_0 = h_1$, $h_1 \in [\![f]\!]_\lambda$. As before, $\mathcal{W}[\![Q_a(v, v)]\!]_\mathcal{A}^{\sigma_l} = \mathtt{Emp}_\mathcal{A}$, so it remains to check that $q_h = \mathcal{W}[\![Q_h(v, v)]\!]_\mathcal{A}^{\sigma_1 \circ \sigma_l} \subseteq \mathtt{Emp}_\mathcal{A}$ and is nonempty. $\sigma_1(\mathtt{x})$ is defined to be $[\![\mathtt{E}]\!]_{\sigma_0}$. As $\sigma_0(\mathtt{x}) = \sigma_l(v)$, $\sigma_1(\mathtt{x}) = [\![\mathtt{E}[v/\mathtt{x}]_{\sigma_l \circ \sigma_1}]\!]$, and so conclude that $q_h = \mathtt{Emp}_\mathcal{A}$ so $h_1 \in [\![q_h * \mathcal{W}[\![Q_a(v, v)]\!]_\mathcal{A}^{\sigma_l} * f]\!]_\lambda$ and therefore D.1.
All the premises of LinPt holds, so we find $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \mathbb{C}) \models_\mathbb{S} \sigma_l, (p_h, p_a, v), \{(q_h, p_a, \langle v, v\rangle)\}$ and finally that $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark) \in [\![\mathbb{S}]\!]$. □

Theorem D.2 (Soundness of MUTATE)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of MUTATE}$$
$$\textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}.$$

Proof.

$$\mathbb{C} = [\mathtt{E}_1] := \mathtt{E}_2$$
$$\mathbb{S} = \mathbb{V}n \in \mathbb{Z}.\ \langle\ fv(\mathtt{E}_1) = \vec{v} \star fv(\mathtt{E}_2) = \vec{w} \mid \mathtt{E}_1[\vec{v}/fv(\mathtt{E}_1)] \mapsto n\ \rangle\ \cdot$$
$$\exists y.\ \langle\ fv(\mathtt{E}_1) = \vec{v} \star fv(\mathtt{E}_2) = \vec{w} \mid \mathtt{E}_1[\vec{v}/fv(\mathtt{E}_1)] \mapsto \mathtt{E}_2[\vec{w}/fv(\mathtt{E}_2)]\ \rangle_{\lambda,\mathcal{A}}$$

Consider $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark) \in [\![\mathbb{C}]\!]_\varphi$, take σ_l, v arbitrary and assume that $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$, where p_h, p_a are as in the definition of $[\![\mathbb{S}]\!]$. In particular, as this implies nonemptiness of the semantics, we find that $\sigma_0(fv(\mathtt{E}_1)) = \sigma_l(\vec{v})$, and $h([\![\mathtt{E}]\!]_{\sigma_0}) = v$, and so from the operational semantics conclude that $\sigma_0 = \sigma_1$ and $h_1 = h_0[[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto [\![\mathtt{E}_2]\!]_{\sigma_0}]$. Furthermore, from Term we find $(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l, (q_h, p_a, \langle v, v\rangle), \{\ (q_h, p_a, \langle v, v\rangle)\ \}$. We aim to apply LinPt to show that $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark)$ is safe. It's clear that $v \in \mathsf{AVal}$. We aim to show

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![Q_a(v, v)]\!]^{\sigma_l}_{\mathcal{A}} \tag{D.2}$$

where $q_h = \mathcal{W}[\![Q_h(v, v)]\!]^{\sigma_1 \circ \sigma_l}_{\mathcal{A}}$. Take $f \in View_{\mathcal{A}}$ and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$. From previous discussion, it's clear that $p_h = \mathtt{Emp}_{\mathcal{A}}$, so in fact $h_0 \in [\![p_a(v) * f]\!]_\lambda$, that is to say, $\exists h, h'.\ h \in [\![p_a(v)]\!]_\lambda$, $h' \in [\![f]\!]_\lambda$ and $h_0 = h \bullet h'$. According to the world semantics, $h = [[\![\mathtt{E}_1[\vec{v}/fv(\mathtt{E}_1)]]\!]_{\sigma_0 \circ \sigma_l} \mapsto [\![v]\!]_{\sigma_l}]$. Given that $h_1 = h_0[[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto [\![\mathtt{E}_2]\!]_{\sigma_0}]$, we may conclude that $h_1 = [[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto [\![\mathtt{E}_2]\!]_{\sigma_0}] \bullet h'$, and thus $h_1 \in [\![\mathcal{W}[\![Q_a(v, v)]\!]^{\sigma_l}_{\mathcal{A}} * f]\!]_\lambda$. Furthermore, as $P_h = Q_h$ and $\sigma_0 = \sigma_1$, $q_h = p_h = \mathtt{Emp}_{\mathcal{A}}$ and $h_1 \in [\![q_h * \mathcal{W}[\![Q_a(v, v)]\!]^{\sigma_l}_{\mathcal{A}} * f]\!]_\lambda$ and we have verified D.2.Given the definition of q_h, we meet all the premises to apply LinPt and obtain $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, S) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{\ (q_h, p_a, \langle v, v\rangle)\ \}$, so $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, S) \in [\![\mathbb{S}]\!]$ and we are done. □

Theorem D.3 (Soundness of CAS)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\textbf{if}\ \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of CAS}$$
$$\textbf{then}\ \models_\Phi \mathbb{C} : \mathbb{S}.$$

Proof.

$$\mathbb{C} = \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3)$$
$$\mathbb{S} = \mathbb{V}n \in \mathbb{Z}.\left\langle \begin{array}{l} \mathtt{x} = x \star \\ \vec{\mathtt{w}} = \vec{w} \end{array} \;\middle|\; \mathtt{E}'_1 \mapsto n \right\rangle \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3)\ \exists y.\left\langle \begin{array}{l} \mathtt{x} = y \star \\ \vec{\mathtt{w}} = \vec{w} \end{array} \;\middle|\; \begin{array}{l} (n = \mathtt{E}'_2 \wedge y = 1 \star \mathtt{E}'_1 \mapsto \mathtt{E}'_3) \vee \\ (n \neq \mathtt{E}'_2 \wedge y = 0 \star \mathtt{E}'_1 \mapsto n) \end{array} \right\rangle_{\lambda,\mathcal{A}}$$

where $\vec{\mathtt{w}} = fv(\mathtt{E}_1) \cup fv(\mathtt{E}_2) \cup fv(\mathtt{E}_3) \setminus \{\mathtt{x}\}$, $\mathtt{E}'_1 = \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]$, $\mathtt{E}'_2 = \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]$ and $\mathtt{E}'_3 = \mathtt{E}_3[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]$. Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by application of CAS, and consider some

$$\tau = (\sigma_0, h_0, \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3))\mathrm{loc}(\sigma_1, h_1, \checkmark) \in [\![\mathbb{C}]\!]_\varphi$$

Take σ_l, v arbitrary and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$, where

$$p_h = \mathcal{W}[\![\mathtt{x} = x \star \vec{\mathtt{w}} = \vec{w}]\!]^{\sigma_0 \circ \sigma_l}_{\mathcal{A}}$$
$$p_a = \lambda n.\ \mathcal{W}[\![\mathtt{E}'_1 \mapsto n \star n \in X]\!]^{\sigma_l}_{\mathcal{A}}$$

Observe the world semantics implies that $[\![\mathtt{E}_1]\!]_{\sigma_0} \in dom(h_0)$. We check two cases corresponding to the applicable rules from the operational semantics.
If $h_0([\![\mathtt{E}_1]\!]_{\sigma_0}) \neq [\![\mathtt{E}_2]\!]_{\sigma_0}$, then $\tau = (\sigma_0, h_0, \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3))\mathrm{loc}(\sigma_1, h_1, \checkmark)$, where $\sigma_1 = \sigma_0[\mathtt{x} \mapsto 0]$ and $h_0 = h_1$. Let $q_h = \mathcal{W}[\![\mathtt{x} = 0 \star \vec{\mathtt{w}} = \vec{w}]\!]^{\sigma_1 \circ \sigma_l}_{\mathcal{A}}$. By Term,

$$(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l, (q_h, p_a, \langle v, 0\rangle), \{(q_h, p_a, \langle v, 0\rangle)\}$$

I aim to show

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![(v = \mathtt{E}'_2 \star 0 = 1 \star \mathtt{E}'_1 \mapsto \mathtt{E}'_3) \vee (v \neq \mathtt{E}'_2 \star 0 = 0 \star \mathtt{E}'_1 \mapsto v)]\!]^{\sigma_l}_{\mathcal{A}}$$

Take $f \in View_{\mathcal{A}}$ arbitrary and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$. From the definition of world semantics, find that some $w_a \in p_a(v), w_f \in f$ such that $h_0 \in \lfloor w_a \bullet w_f \rfloor_\lambda$, $\sigma_0(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$ and $\sigma_0(\mathtt{x}) = \sigma_l(x)$. Therefore, from the operational semantics, we find $\sigma_0(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$ and $\sigma_0(\mathtt{x}) = 0 = \sigma_1(v)$, so $q_h = \mathtt{Emp}_{\mathcal{A}}$ and for some $w_h \in q_h$, $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_\lambda$. It remains to show $w_a \in q_a$, where

$$q_a = \mathcal{W}[\![(v = \mathtt{E}'_2 \wedge y = 1 \star \mathtt{E}'_1 \mapsto \mathtt{E}'_3) \vee (v \neq \mathtt{E}'_2 \wedge y = 0 \star \mathtt{E}'_1 \mapsto v)]\!]^{\sigma_l}_{\mathcal{A}}$$

Well, $p_a(v)$ tells us that $h_0([\![\mathtt{E}_1]\!]\sigma_0) = v$, and the initial assumption that $h_0([\![\mathtt{E}_2]\!]\sigma_0) \neq v$, so given that $\mathcal{W}[\![v \neq \mathtt{E}_2' \star 0 = 0]\!]^{\sigma_l}_{\mathcal{A}}$ is nonempty,

$$\mathcal{W}[\![\mathtt{E}_1' \mapsto v]\!]^{\sigma_l}_{\mathcal{A}} = \mathcal{W}[\![v \neq \mathtt{E}_2' \star 0 = 0 \star \mathtt{E}_1' \mapsto v]\!]^{\sigma_l}_{\mathcal{A}} \subseteq q_a$$

and $w_a \in \mathcal{W}[\![\mathtt{E}_1' \mapsto v]\!]^{\sigma_l}_{\mathcal{A}}$, so $h_0 \in [\![q_h * q_a * f]\!]_\lambda$. Conclude by application of LinPt that

$$(\sigma_0, h_0, \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3))\mathrm{loc}(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l(p_h, p_a, v), \{(q_h, p_a, \langle v, 0\rangle)\}$$

If $h_0([\![\mathtt{E}_1]\!]_{\sigma_0}) \neq [\![\mathtt{E}_2]\!]_{\sigma_0}$, then from the operational semantics, it must be that

$$\tau = (\sigma_0, h_0, \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3))\mathrm{loc}(\sigma_1, h_1, \checkmark)$$

where $\sigma_1 = \sigma_0[\mathtt{x} \mapsto 1]$, $h_1 = h_0[[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto [\![\mathtt{E}_3]\!]_{\sigma_0}]$. Again, let $q_h = \mathcal{W}[\![\mathtt{x} = 1 \star \vec{\mathtt{w}} = \vec{w}]\!]^{\sigma_1 \circ \sigma_l}_{\mathcal{A}}$, and then by Term, $(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l, (q_h, p_a, \langle v, 1\rangle), \{(q_h, p_a, \langle v, 1\rangle)\}$. We aim to check

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![(v = \mathtt{E}_2' \star 1 = 1 \star \mathtt{E}_1' \mapsto \mathtt{E}_3') \vee (v \neq \mathtt{E}_2' \star 1 = 0 \star \mathtt{E}_1' \mapsto v)]\!]^{\sigma_l}_{\mathcal{A}}$$

Take $f \in View_{\mathcal{A}}$, assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$, and therefore find such $w_h \in p_h, w_a \in p_a(v), w_f \in f$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_\lambda$. The implied nonemptiness of p_h tells us that $\sigma_0(\mathtt{x}) = \sigma_l(x)$ and $\sigma_0(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$, and therefore that $\sigma_0(\mathtt{x}) = 1$ and $\sigma_1(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$. So $q_h = \mathtt{Emp}_{\mathcal{A}}$, and furthermore $w_h \in q_h$. The definition of p_a tells us that $w_a = (h_a, \_, \_, \_)$ is such that $h_a = [[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto v]$, and the premise of the operational semantics that $[\![\mathtt{E}_2]\!]_{\sigma_0} = v$ and therefore the definition of h_1 makes it clear that for $w_a' = (h_a', \_, \_, \_)$ such that $h_a' = [[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto [\![\mathtt{E}_3]\!]_{\sigma_0}]$, $h_1 \in \lfloor w_h \bullet w_a' \bullet w_f \rfloor_\lambda$ (as the disjointness of the heaps in world composition implies the change between h_0 and h_1 can't affect the heaps of w_h or w_f). Let $q_a = \mathcal{W}[\![(v = \mathtt{E}_2' \star 1 = 1 \star \mathtt{E}_1' \mapsto \mathtt{E}_3') \vee (v \neq \mathtt{E}_2' \star 1 = 0 \star \mathtt{E}_1' \mapsto v)]\!]^{\sigma_l}_{\mathcal{A}}$. Then

$$\mathcal{W}[\![v = \mathtt{E}_2' \star 1 = 1 \star \mathtt{E}_1' \mapsto \mathtt{E}_3']\!]^{\sigma_l}_{\mathcal{A}} \subseteq \mathcal{W}[\![(v = \mathtt{E}_2' \star 1 = 1 \star \mathtt{E}_1' \mapsto \mathtt{E}_3') \vee (v \neq \mathtt{E}_2' \star 1 = 0 \star \mathtt{E}_1' \mapsto v)]\!]^{\sigma_l}_{\mathcal{A}}$$

Obviously $w_a' \in \mathcal{W}[\![\mathtt{E}_1' \mapsto \mathtt{E}_3']\!]^{\sigma_l}_{\mathcal{A}}$. We know that $v = [\![\mathtt{E}_2]\!]_{\sigma_0}$, and with the replacements of the logical variables, it remains the case that $\mathcal{W}[\![v = \mathtt{E}_2' \star 1 = 1]\!]^{\sigma_l}_{\mathcal{A}} = \mathtt{Emp}_{\mathcal{A}}$, so $w_a' \in \mathcal{W}[\![v = \mathtt{E}_2' \star 1 = 1 \star \mathtt{E}_1' \mapsto \mathtt{E}_3']\!]^{\sigma_l}_{\mathcal{A}}$. Conclude that $h_1 \in [\![q_h * q_a * f]\!]_\lambda$, so by LinPt,

$$(\sigma_0, h_0, \mathtt{x} := \mathtt{CAS}(\mathtt{E}_1, \mathtt{E}_2, \mathtt{E}_3))\mathrm{loc}(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l(p_h, p_a, v), \{(q_h, p_a, \langle v, 1\rangle)\}$$

and conclude in both cases that $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. □

Theorem D.4 (Soundness of FAS)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

if $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by application of FAS
then $\models_\Phi \mathbb{C} : \mathbb{S}$.

Proof.

$$\begin{aligned}
&\mathbb{C} = \mathtt{x} := \mathtt{FAS}(\mathtt{E}_1, \mathtt{E}_2)\\
&\mathbb{S} = \forall n \in \mathbb{Z}. \left\langle \begin{array}{l} \mathtt{x} = x \star \\ \vec{\mathtt{w}} = \vec{w} \end{array} \;\middle|\; \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \right\rangle \mathtt{x} := \mathtt{FAS}(\mathtt{E}_1, \mathtt{E}_2)\\
&\qquad\qquad \left\langle \begin{array}{l} \mathtt{x} = n \star \\ \vec{\mathtt{w}} = \vec{w} \end{array} \;\middle|\; \mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \right\rangle_{\lambda,\mathcal{A}}
\end{aligned}$$

where $\vec{\mathtt{w}} = fv(\mathtt{E}_1) \cup fv(\mathtt{E}_2) \setminus \{\mathtt{x}\}$.
Consider $\tau \in [\![\mathbb{C}]\!]_\varphi$, where the first step is a local step, take σ_l, v arbitrary and assume

$$h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$$

where

$$\begin{aligned}
p_h &= \mathcal{W}[\![\mathtt{x} = x \star \vec{\mathtt{w}} = \vec{w}]\!]^{\sigma_0 \circ \sigma_l}_{\mathcal{A}}\\
p_a &= \lambda n.\ \mathcal{W}[\![\mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto n \star n \in X]\!]^{\sigma_l}_{\mathcal{A}}
\end{aligned}$$

From the definition of world semantics, it must be that $\sigma_0(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$ and $\sigma_0(\mathtt{x}) = \sigma_l(x)$, so $h_0([\![\mathtt{E}_1]\!]) = v$. Therefore, the operational semantics tells us that $\tau = (\sigma_0, h_0, \mathbb{C}_0)\mathrm{loc}(\sigma_0, h_0, \checkmark)$, $\sigma_1 = \sigma_0[\mathtt{x} \mapsto h([\![\mathtt{E}_1]\!]_{\sigma_0})]$ and $h_1 = h_0[[\![\mathtt{E}_1]\!]_{\sigma_0} \mapsto [\![\mathtt{E}_2]\!]_{\sigma_0}]$. Let $q_h = \mathcal{W}[\![\mathtt{x} = n \star \vec{\mathtt{w}} = \vec{w}]\!]^{\sigma_1 \circ \sigma_l}_{\mathcal{A}}$. Term tells us that

$$(\sigma_0, h_0, \checkmark) \models_{\mathbb{S}} \sigma_l, (q_h, p_a, \langle v, v\rangle), \{(q_h, p_a, \langle v, v\rangle)\}$$

We aim to prove

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![\mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]]\!]^{\sigma_l}_{\mathcal{A}}$$

Take $f \in View_{\mathcal{A}}$ and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$, i.e. find some $w_h \in p_h, w_a \in p_a(v), w_f \in f$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_\lambda$. In particular, the nonemptiness of the world semantics imply that $\sigma_0(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$ and $\sigma_0(\mathtt{x}) = \sigma_l(x)$, and so $\sigma_1(\vec{\mathtt{w}}) = \sigma_l(\vec{w})$ and $\sigma_1(\mathtt{x}) = v$. From w_a, we find that $h_0([\![\mathtt{E}_1]\!]_{\sigma_0\circ\sigma_l}) = v$, so $h_1([\![\mathtt{E}_1]\!]_{\sigma_0\circ\sigma_l}) = [\![\mathtt{E}_2]\!]_{\sigma_0\circ\sigma_l}$. From all this we can conclude $w_h \in q_h$ and find some $w'_a \in \mathcal{W}[\![\mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]]\!]^{\sigma_l}_{\mathcal{A}}$ which differs only from w_a in the update of the heap component. Thus $h_1 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_\lambda$ by considering the disjointness requirements on heap components, and so

$$h_1 \in [\![q_h * \mathcal{W}[\![\mathtt{E}_1[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}] \mapsto \mathtt{E}_2[\vec{w}/\vec{\mathtt{w}}, x/\mathtt{x}]]\!]^{\sigma_l}_{\mathcal{A}} * f]\!]_\lambda$$

Conclude by an application of LinPt that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(q_h, p_a, \langle v, v\rangle)\}$ □

Theorem D.5 (Soundness of DEALLOC)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\begin{array}{l}\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of DEALLOC} \\ \quad \textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}.\end{array}$$

Proof. Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by an application of Dealloc, that is,

$$\mathbb{S} = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ \exists z.\ \mathtt{E} \mapsto z \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ \mathsf{emp} \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}}$$

Consider $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}C_1 \in [\![\mathbb{C}]\!]_\varphi$. Take σ_l, v arbitrary and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_\lambda$, where $p_h = \mathcal{W}[\![\exists z.\ \mathtt{E} \mapsto z]\!]^{\sigma_0\circ\sigma_l}_{\mathcal{A}}$ and $p_a = \lambda x.\ \mathtt{Emp}_{\mathcal{A}}$. The definition of world semantics of p_h tells us that $[\![\mathtt{E}]\!]_{\sigma_0} \in dom(h_0)$ and therefore from the operational semantics find that $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark)$, where $\sigma_0 = \sigma_1$ and $h_1 = h_0[[\![\mathtt{E}]\!]_{\sigma_0} \mapsto \bot]$. Let $q_h = q_a = \mathtt{Emp}_{\mathcal{A}}$, then clearly by Term, $(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l, (q_h, p_a, \langle v, v\rangle), \{(q_h, p_a, \langle v, v\rangle)\}$. I aim to prove

$$\begin{array}{rl}(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) & \implies q_h * q_a \\ & q_h = \mathcal{W}[\![emp]\!]^{\sigma_1\circ\sigma_l}_{\mathcal{A}}\end{array}$$

The second is true by definition. Take $f \in View_{\mathcal{A}}$ arbitrary and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$. From the definition of reification, find $\exists w_h \in p_h, w_f = (h_f, \_, \_, \_) \in f$ such that $h_0 \in \lfloor w_h \bullet w_f \rfloor_\lambda$ (as $p_a(v) = \mathtt{Emp}_{\mathcal{A}}$). Furthmore, from the definition of p_h, find that $\exists z.\ w_h = ([[\![\mathtt{E}]\!]_{\sigma_0} \mapsto z], \rho, \gamma, \chi)$, and $(\bot, \rho, \gamma, \chi) \in \mathtt{Emp}_{\mathcal{A}}$, and that from world composition, $[\![\mathtt{E}]\!]_{\sigma_0} \notin dom(h_f)$. Therefore, as $h_1 = h_0[[\![\mathtt{E}]\!]_{\sigma 0} \mapsto \bot]$, $h_1 \in \lfloor (\bot, \rho, \gamma, \chi) \bullet w_f \rfloor_\lambda$, and conclude $h_1 \in [\![q_h * q_a * f]\!]_\lambda$. This proves precisely the first equation. So we have verified the premises of LinPt, and conclude that $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \checkmark) \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(q_h, q_a, \langle v, v\rangle)\}$. □

D.2 Hoare

Theorem D.6 (Soundness of IF)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\begin{array}{l}\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of IF} \\ \quad \textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}.\end{array}$$

Proof.

$$\begin{array}{l}\mathbb{C} = \mathtt{if}\ (\mathtt{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2 \\ \mathbb{S} = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}}\end{array}$$

Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$. Then for

$$\begin{array}{l}\mathbb{S}_1 = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ P \star \mathtt{B} \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}} \\ \mathbb{S}_2 = \mathbb{V}x \in \mathsf{AVal}.\ \langle\ P \star \neg\mathtt{B} \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ Q \mid \mathsf{emp}\rangle_{\lambda,\mathcal{A}}\end{array}$$

we have $\vdash_\Phi \mathbb{C}_1 : \mathbb{S}_1$ and $\vdash_\Phi \mathbb{C}_2 : \mathbb{S}_2$. By the inductive hypothesis we have

$$\models_\Phi \mathbb{C}_1 : \mathbb{S}_1 \tag{D.3}$$

$$\models_\Phi \mathbb{C}_2 : \mathbb{S}_2 \tag{D.4}$$

We need to prove $\models_\Phi \mathbb{C} : \mathbb{S}$, so take φ arbitrary and assume that $\models \varphi : \Phi$. Take $\tau \in [\![\mathbb{C}]\!]_\varphi$ arbitrary, $\sigma_l \in \mathsf{LStore}$, $v \in \mathsf{AVal}$, and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_\mathcal{A}]\!]_\lambda$. As usual, if τ is a single state, an application of Term gives the necessary safety result. For any other τ, $\tau \in [\![\mathbb{C}_\varphi]\!]$ gives $fv(\mathtt{B}) \subseteq dom(\sigma_0)$ and so the applicable first operational semantics steps are

$$\frac{[\![\mathtt{B}]\!]_\sigma}{(\sigma, h, \mathtt{if}\ (\mathtt{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2) \xrightarrow{\mathrm{loc}}_\varphi (\sigma, h, \mathbb{C}_1)} (1) \qquad \frac{\neg[\![\mathtt{B}]\!]_\sigma}{(\sigma, h, \mathtt{if}\ (\mathtt{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2) \xrightarrow{\mathrm{loc}}_\varphi (\sigma, h, \mathbb{C}_2)} (2)$$

Otherwise, $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau'$, where $\sigma_0 = \sigma_1$, $h_0 = h_1$, $(\sigma_1, h_1, \mathbb{C}_1)\tau' \in [\![\mathbb{C}_1]\!]'$. If $\tau \in \mathsf{Trace}$ by rule (1), then the premise of (1) and assumption on h_0, $h_1 \in [\![\mathcal{W}[\![P \star \mathtt{B}]\!]_\mathcal{A}^{\sigma_1 \circ \sigma_l} * p_a(v) * \mathrm{True}_\mathcal{A}]\!]_\lambda$, thus the inductive hypothesis finds some S such that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}_1} \sigma_l, (\mathcal{W}[\![P \star \mathtt{B}]\!]_\mathcal{A}^{\sigma_1 \circ \sigma_l}, p_a, v), S$ As $\mathbb{S}_1$ and $\mathbb{S}$ agree on all but the precondition, find also that $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_\mathbb{S} \sigma_l, (q_h, p_a, v), S$, where $q_h = \mathcal{W}[\![P \star \mathtt{B}]\!]_\mathcal{A}^{\sigma_1 \circ \sigma_l}$. Therefore I need to check

$$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * p_a(v) \tag{D.5}$$

Take $f \in View_\mathcal{A}$ arbitrary and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$. As the atomic part of the condition is empty, $p_a(v) = \mathtt{Emp}_\mathcal{A}$, so in fact $\exists h, h'.\ h_0 = h \bullet h'$, $h \in [\![p_h]\!]_\lambda$ and $h' \in [\![f]\!]_\lambda$. Given the premise of the operational semantic rule, it must be that $[\![\mathtt{B}]\!]_{\sigma_0}$ is true. Therefore $\mathcal{W}[\![\mathtt{B}]\!]_\mathcal{A}^{\sigma_0 \circ \sigma_l} = \mathtt{Emp}_\mathcal{A}$ and $h \in [\![p_h * \mathcal{W}[\![\mathtt{B}]\!]_\mathcal{A}^{\sigma_0 \circ \sigma_l}]\!]_\lambda = [\![p'_h]\!]_\lambda$. As $h_0 = h_1$, find that $h_1 = h \bullet h' \in [\![p'_h * p_a(v) * f]\!]_\lambda$ and thus D.5 holds. We know $\mathbb{C}_1 \neq \checkmark$, so by Stutter, $(\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_\mathbb{S} \sigma_l, (p_h, p_a, v), S$ and is in $[\![\mathbb{S}]\!]$. The case for the second operational semantic rule is completely analagous. □

Theorem D.7 (Soundness of WHILE)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

if $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by application of WHILE
then $\models_\Phi \mathbb{C} : \mathbb{S}$.

Proof.

$$\mathbb{C} = \mathtt{while}\ (\mathtt{B})\ \mathbb{C}'$$
$$\mathbb{S} = \forall x \in \mathsf{AVal}.\ \langle\ P \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ P \star \neg \mathtt{B} \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$. Then for

$$\mathbb{S}' = \forall x \in \mathsf{AVal}.\ \langle\ P \star \mathtt{B} \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ P \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

we have $\vdash_\Phi \mathbb{C}' : \mathbb{S}'$ and by the inductive hypothesis we have

$$\models_\Phi \mathbb{C}' : \mathbb{S}'$$

We need to prove $\models_\Phi \mathbb{C} : \mathbb{S}$, so take φ arbitrary and assume that $\models \varphi : \Phi$. Take $\tau \in [\![\mathbb{C}]\!]_\varphi$ arbitrary, $\sigma_l \in \mathsf{LStore}$, $v \in \mathsf{AVal}$, and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_\mathcal{A}]\!]_\lambda$. I have the following recurrence relation for τ

$$(\neg[\![\mathtt{B}]\!]_{\sigma_0} \wedge \tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_0, h_0, \checkmark)) \vee$$
$$[\![\mathtt{B}]\!]_{\sigma_0} \wedge \tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_0, h_0, \mathbb{C}'\ ;\ \mathbb{C})\tau' \wedge (\sigma_0, h_0, \mathbb{C}'\ ;\ \mathbb{C})\tau' \in [\![\mathbb{C}']\!]_\varphi ; [\![\mathbb{C}]\!]_\varphi$$

from the operational semantics and Lemma 6.3. So we proceed by induction on how many times τ is decomposable by the second case. If none, it must be that the first case applies, and thus $\neg[\![\mathtt{B}]\!]_{\sigma_0} \wedge \tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_0, h_0, \checkmark)$. We can use Term to deduce that $(\sigma_0, h_0, \checkmark) \models_\mathbb{S} \sigma_l, (q_h, p_a, v), \{(q_h, p_a, v)\}$, where $q_h = \mathcal{W}[\![P \star \neg \mathtt{B}]\!]_\mathcal{A}^{\sigma_0 \star \sigma_0}$. An application of LinPt will give us $\tau \models_\mathbb{S} \sigma_l, (p_h, p_a, v), \{(q_h, p_a, v)\}$ so we check the rest of the premises:
$(h_0, h_0) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathtt{Emp}_\mathcal{A}$ is given by $p_a(v) = \mathtt{Emp}_\mathcal{A}$ and $\mathcal{W}[\![P]\!]_\mathcal{A}^{\sigma_0 \star \sigma_0} = \mathcal{W}[\![P \star \neg \mathtt{B}]\!]_\mathcal{A}^{\sigma_0 \star \sigma_0}$, as $\neg[\![\mathtt{B}]\!]_{\sigma_0} \implies \mathcal{W}[\![\neg \mathtt{B}]\!]_\mathcal{A}^{\sigma_0 \star \sigma_l} = \mathtt{Emp}_\mathcal{A}$. We end in $\checkmark$ so we must check q_h is the postcondition but this is true by construction, we conclude with $\tau \in [\![\mathbb{S}]\!]$.

In the second case, $\tau = (\sigma_0, h_0, \mathbb{C})\mathrm{loc}(\sigma_0, h_0, \mathbb{C}'\ ;\ \mathbb{C})\tau' \wedge (\sigma_0, h_0, \mathbb{C}'\ ;\ \mathbb{C})\tau' \in [\![\mathbb{C}']\!]_\varphi ; [\![\mathbb{C}]\!]_\varphi$. We have from our inductive hypothesis and from Lemma 6.5 that

$$\exists S.\ (\sigma_0, h_0, \mathbb{C}'\ ;\ \mathbb{C})\tau' \models_{\mathbb{S}''} \sigma_l, (p'_h, p_a, v), S$$

where $p'_h = \mathcal{W}[\![P \star \mathtt{B}]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$ and

$$\mathbb{S}'' \triangleq \forall x \in \mathsf{AVal}.\ \langle\ P \star \mathtt{B} \mid \mathsf{emp}\ \rangle\ \cdot\ \exists y.\ \langle\ P \star \neg \mathtt{B} \mid \mathsf{emp}\ \rangle_{\lambda,\mathcal{A}}$$

It remains therefore to deduce that for the same S

$$(\sigma_0, h_0, \mathbb{C}'\ ;\ \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p'_h, p_a, v), S$$

by a trivial induction on the trace safety judgement ($\mathbb{S}''$ and $\mathbb{S}$ agree on all components except the Hoare precondition which does not occur in the trace safety judgement, so all premises of each rule are immediately transferable). To apply Stutter, clearly $\mathbb{C}'\ ;\ \mathbb{C} \neq \checkmark$, so it remains to check that $(h_0, h_0) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow p'_h * p_a(v)$, but in fact as by assumption $[\![\mathtt{B}]\!]_{\sigma_0}$, we find that $\mathcal{W}[\![\mathtt{B}]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l} = \mathtt{Emp}_{\mathcal{A}}$ and therefore $p_h = p'_h$. So Stutter tells us $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ and finally $\tau \in [\![\mathbb{S}]\!]$. □

D.3 Logical

Theorem D.8 (Soundness of ATOMIC-WEAKENING)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\textbf{if}\ \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of ATOMIC WEAKENING}$$
$$\textbf{then}\ \models_\Phi \mathbb{C} : \mathbb{S}.$$

Proof. Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by an application of Atomic Weakening, that is, for

$$\mathbb{S} = \forall x \in X.\langle\ P_h \star P \mid P_a(x)\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \star Q(x,y) \mid Q_a(x,y)\ \rangle_{\lambda,\mathcal{A}}$$
$$\mathbb{S}' = \forall x \in X.\langle\ P_h \mid P_a(x) \star P\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid Q_a(x,y) \star Q(x,y)\ \rangle_{\lambda,\mathcal{A}}$$

from the premises of the rule we have $\vdash_\Phi \mathbb{C} : \mathbb{S}'$, $\mathcal{A} \models P_h \star P$ stable and $\forall x, y,\ \mathcal{A} \models Q(x,y)$ stable.

Lemma D.1.
$\forall h_0, h_1 \in \mathsf{Heap}, \tau \in \mathsf{Trace}, p'_h \in View_{\mathcal{A}}, \sigma_l \in \mathsf{LStore}, v \in \mathsf{AVal}', S.\ \tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S' \implies \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ where

$$p_a = \lambda x.\ \begin{cases} \mathcal{W}[\![P_a(x) \star x \in X]\!]_{\mathcal{A}}^{\sigma_l} & x \in \mathsf{AVal} \\ \mathtt{Emp}_{\mathcal{A}} & \text{otherwise} \end{cases}$$
$$p'_a = \lambda x.\ \begin{cases} \mathcal{W}[\![P_a(x) \star P \star x \in X]\!]_{\mathcal{A}}^{\sigma_l} & x \in \mathsf{AVal} \\ \mathtt{Emp}_{\mathcal{A}} & \text{otherwise} \end{cases}$$
$$p_h = \begin{cases} p'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} & v \in \mathsf{AVal} \\ p'_h * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} & \text{otherwise} \end{cases}$$
$$S \subseteq \left\{ (q_h, p_a, v') \;\middle|\; (q'_h, p'_a, v') \in S' \wedge q_h = \begin{cases} q'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} & v' \in \mathsf{AVal} \\ q'_h * \mathcal{W}[\![Q(v')]\!]_{\mathcal{A}}^{\sigma_l} & \text{otherwise} \end{cases} \right\}$$

Proof. Take $\forall h_0, h_1 \in \mathsf{Heap}, \tau \in \mathsf{Trace}, p'_h \in View_{\mathcal{A}}, \sigma_l \in \mathsf{LStore}, v \in \mathsf{AVal}', S$ arbitrary and assume $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$. Proceed by induction on the trace safety judgement.

- **Case: Term** If $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$ is by an application of Term then it must be that $\tau = (\sigma, h, \mathbb{C})$, and $S' = \{(p'_h, p'_a, v)\}$. Let

  $$p_h = \begin{cases} p'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} & v \in \mathsf{AVal} \\ p'_h * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} & \text{otherwise} \end{cases}$$

  By application of Term, $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \{(p_h, p_a, v)\}$ which satisfies the necessary properties.

- **Case: Stutter** If $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$ is by an application of Stutter then it must be that $\tau = (\sigma_0, h_0, \mathbb{C}_0)\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau'$, $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (q'_h, p'_a, v), S'$, $(h_0, h_1) \models_{\lambda,\mathcal{A}} p'_h * p'_a(v) \twoheadrightarrow q'_h * p'_a(v)$

and $\mathbb{C}_1 = \checkmark \implies q'_h = \mathcal{W}[\![Q_h(v)]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l} \wedge v \in \mathsf{AVal} \times \mathsf{AVal}$. The inductive hypothesis tells us that $\exists S.\ (\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (q_h, p'_a, v), S$, the necessary relation between S' and S holds, and

$$q_h = \begin{cases} q'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} & v \in \mathsf{AVal} \\ q'_h * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} & \text{otherwise} \end{cases}$$

We aim to prove that for

$$p_h = \begin{cases} p'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} & v \in \mathsf{AVal} \\ p'_h * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} & \text{otherwise} \end{cases}$$

$(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * p_a(v)$ and $\mathbb{C}_1 = \checkmark \implies q_h = \mathcal{W}[\![Q_h(v) \star Q(v)]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l} \wedge v \in \mathsf{AVal} \times \mathsf{AVal}$.
In the second case, if $\mathbb{C}_1 = \checkmark$ then from the premises of Stutter we have $q'_h = \mathcal{W}[\![Q_h(v)]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l} \wedge v \in \mathsf{AVal} \times \mathsf{AVal}$. Thus $q_h = q'_h * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} = \mathcal{W}[\![Q_h(v) \star Q(v)]\!]_{\mathcal{A}}^{\sigma_l} \wedge v \in \mathsf{AVal} \times \mathsf{AVal}$ holds.
Take $f \in View_{\mathcal{A}}$ and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$, i.e. find some $w_h \in p_h, w_a \in p_a(v), w_f \in f$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_\lambda$. If $v \in \mathsf{AVal}$, then we can find some $w'_h \in p'_h, w_p \in \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l}$ such that $w_h = w'_h \bullet w_p$ and as $w_a \bullet w_p \in p'_a(v)$, conclude that $h_0 \in [\![p'_h * p'_a(v) * f]\!]_\lambda$. From the premise of Stutter, find that $h_1 \in [\![q'_h * p'_a(v) * f]\!]_\lambda$. From the assumption on q_h and q'_h, similar rearrangement finds that $h_1 \in [\![q_h * p_a(v) * f]\!]_\lambda$.
If $v \in \mathsf{AVal} \times \mathsf{AVal}$, then find some $w'_h \in p'_h, w_q \in \mathcal{W}[\![Q(x,y)]\!]_{\mathcal{A}}^{\sigma_l}$ such that $w_h = w'_h \bullet w_q$ and $h_0 \in [\![p'_h * p'_a(v) * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} * f]\!]_\lambda$ (as $p'_a(v) = \mathtt{Emp}_{\mathcal{A}} = p_a(v)$). As $Q(v)$ is stable, from the premise on Stutter find that $h_1 \in [\![q'_h * p'_a(v) * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} * f]\!]_\lambda$ and similar rearrangement from the definitions of q_h and q'_h, $p_a(v)$ and $p'_a(v)$ gives $h_1 \in [\![q_h * p_a(v) * f]\!]_\lambda$.
So we have checked everything we need to apply Stutter to τ and conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$.

- **Case: LinPt** If $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$ is by an application of LinPt then it must be that $\tau = (\sigma_0, h_0, \mathbb{C}_0)\mathrm{loc}(\sigma_1, h_1, \mathbb{C}_1)\tau'$, $(\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (q'_h, p'_a, \langle v, v' \rangle), S'$, $(h_0, h_1) \models_{\lambda,\mathcal{A}} p'_h * p'_a(v) \twoheadrightarrow q'_h * \mathcal{W}[\![Q_a(v,v') \star Q(v,v')]\!]_{\mathcal{A}}^{\sigma_l}$, $v \in \mathsf{AVal}$ and $\mathbb{C}_1 = \checkmark \implies q'_h = \mathcal{W}[\![Q_h(v,v')]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$.
The inductive hypothesis tells us that $\exists S.\ (\sigma_1, h_1, \mathbb{C}_1)\tau' \models_{\mathbb{S}'} \sigma_l, (q_h, p'_a, \langle v, v' \rangle), S$, the necessary relation between S' and S holds, and $q_h = q'_h * \mathcal{W}[\![Q(v,v')]\!]_{\mathcal{A}}^{\sigma_l}$. We aim to prove that for $p_h = p'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l}$, $(h_0, h_1) \models_{\lambda,\mathcal{A}} p_h * p_a(v) \twoheadrightarrow q_h * \mathcal{W}[\![Q_a(v,v')]\!]_{\mathcal{A}}^{\sigma_l}$ and $\mathbb{C}_1 = \checkmark \implies q_h = \mathcal{W}[\![Q_h(v,v') \star Q(v,v')]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$.
In the second case, if $\mathbb{C}_1 = \checkmark$ then from the premises of Stutter we have $q'_h = \mathcal{W}[\![Q_h(v,v')]\!]_{\mathcal{A}}^{\sigma_1 \circ \sigma_l}$. Thus $q_h = q'_h * \mathcal{W}[\![Q(v)]\!]_{\mathcal{A}}^{\sigma_l} = \mathcal{W}[\![Q_h(v,v') \star Q(v,v')]\!]_{\mathcal{A}}^{\sigma_l}$ holds.
Take $f \in View_{\mathcal{A}}$ and assume $h_0 \in [\![p_h * p_a(v) * f]\!]_\lambda$, i.e. find some $w_h \in p_h, w_a \in p_a(v), w_f \in f$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_\lambda$. We can find some $w'_h \in p'_h, w_p \in \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l}$ such that $w_h = w'_h \bullet w_p$ and as $w_a \bullet w_p \in p'_a(v)$, conclude that $h_0 \in [\![p'_h * p'_a(v) * f]\!]_\lambda$. From the premise of LinPt, find that $h_1 \in [\![q'_h * \mathcal{W}[\![Q_a(v,v') \star Q(v,v')]\!]_{\mathcal{A}}^{\sigma_l} * f]\!]_\lambda$. From the assumption on q_h and q'_h, similar rearrangement finds that $h_1 \in [\![q_h * \mathcal{W}[\![Q_a(v,v')]\!]_{\mathcal{A}}^{\sigma_l} * f]\!]_\lambda$.
So we have checked everything we need to apply LinPt to τ and conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$.

- **Case: Env** If $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$ is by an application of Env', then $\tau = (\sigma, h_0, \mathbb{C})\mathrm{env}(\sigma, h_1, \mathbb{C})\tau'$, $v \in \mathsf{AVal}$ and

$$\forall v' \in X.\ E(v') \implies (\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v'), S'_{v'} \tag{D.6}$$

, where $E(v') = \exists p_e, p'_e.\ h_0 \in [\![p'_h * p'_a(v) * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p'_a(v) * p_e \twoheadrightarrow p'_a(v') * p'_e$, and $S' = \bigcup_{v' \in X.\ E(v')} S'_{v'}$.
Let $p_h = p'_h * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l}$, take $v' \in X$ arbitrary and find p_e, p'_e such that

$$h_0 \in [\![p_h * p_a(v) * p_e]\!]_\lambda \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_a(v) * p_e \twoheadrightarrow p_a(v') * p'_e \tag{D.7}$$

I aim to show this implies $E(v')$ for the same p_e, p'_e. Find $w_h \in p_h, w_a \in p_a(v), w_e \in p_e$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_e \rfloor_\lambda$, and furthermore from the definition of p_h, find some $w'_h \in p'_h, w_p \in \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l}$ such that $w_h = w'_h \bullet w_p$. Then conclude that $w_a \bullet w_p \in p'_a(v)$, and therefore $h_0 \in [\![p'_h * p'_a(v) * p_e]\!]_\lambda$.
To show the second conjunct, take $f \in View_{\mathcal{A}}$ arbitrary and assume $h_0 \in [\![p'_a(v) * p_e * f]\!]_\lambda$. Then $\exists w'_a \in p'_a(v), w_e \in p_e, w_f \in f$ such that $h_0 \in \lfloor w'_a \bullet w_e \bullet w_f \rfloor_\lambda$, and from the definition of p'_a it's clear that w'_a can be further subdivided into some $w_a \in p_a(v)$ and $w_p \in \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l}$ such that $h_0 \in [\![p_a(v) * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} * p_e * f]\!]_\lambda$. As P is $\mathcal{A}$-stable, from the assumption, find that $h_1 \in [\![p_a(v') * \mathcal{W}[\![P]\!]_{\mathcal{A}}^{\sigma_l} * p_e * f]\!]_\lambda$ and similar reasoning about worlds allows us to conclude that $h_1 \in [\![p'_a(v') * p_e * f]\!]_\lambda$. So we have shown our assumption implies $E(v')$.
Therefore, for all $v' \in X$ such that D.7 holds, $E(v')$ also holds, so from the premise of Env it

must be that $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v'), S'_{v'}$ and finally from the inductive hypothesis that $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p'_a, v'), S_{v'}$ for appropriate p_h and $S_{v'}$. Conclude that $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$, $S \subseteq \bigcup_{v' \in X.\ E(v')} S_{v'}$ and we have the result.

- **Case: Env'** If $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$ is by an application of Env', then $\tau = (\sigma, h_0, \mathbb{C})\text{env}(\sigma, h_1, \mathbb{C})\tau'$, $v \in \mathsf{AVal} \times \mathsf{AVal}$, and if $\exists p_e, p'_e.\ h_0 \in [\![p'_h * p_e]\!] \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e$ then $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$, else $S' = \emptyset$. As $v \in \mathsf{AVal} \times \mathsf{AVal}$, let $p_h = p'_h * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}}$. Let's assume $\exists p_e, p'_e.\ h_0 \in [\![p_h * p_e]\!] \wedge (h_0, h_1) \models_{\lambda,\mathcal{A}} p_e \twoheadrightarrow p'_e$. Then clearly $h_0 \in [\![p'_h * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}} * p_e]\!]_\lambda$, so by taking $\overline{p}_e = p_e * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}}$ find that $h_0 \in [\![p'_h * \overline{p}_e]\!]_\lambda$. Similarly, by taking $\overline{p}'_e = p'_e * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}}$, and $f \in View_{\mathcal{A}}$, $h_0 \in [\![\overline{p}_e * f]\!]_\lambda \implies h_0 \in [\![p_e * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}} * f]\!]_\lambda$ by definition, and from the stability of $Q(v)$ and assumption find $h_1 \in [\![p'_e * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}} * f]\!]_\lambda$ and finally again the definition of $\overline{p}'_e$ yields $h_1 \in [\![\overline{p}'_e * f]\!]_\lambda$. Therefore from the premise of Env' and the inductive hypothesis, find $(\sigma, h_1, \mathbb{C})\tau' \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$, for S satisfying the right relation to S', so by Env' $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$. Furthermore, if the initial assumption doesn't hold, then we can freely conclude $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \emptyset$.

- **Case: Env$\lightning$** If $\tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$ is by an application of Env$\lightning$, then it must be the case that $\tau = (\sigma, h, \mathbb{C})\text{env}\lightning\,\tau'$, $S' = \emptyset$, and therefore $\tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), \emptyset$ for

$$p_h = \begin{cases} p'_h * \mathcal{W}[\![P]\!]^{\sigma_l}_{\mathcal{A}} & v \in \mathsf{AVal} \\ p'_h * \mathcal{W}[\![Q(v)]\!]^{\sigma_l}_{\mathcal{A}} & \text{otherwise} \end{cases}$$

holds by Env$\lightning$ and satisfies our requirements.

□

From the inductive hypothesis, we find that $\models_\Phi \mathbb{C} : \mathbb{S}'$, i.e. $[\![\mathbb{C}]\!]_\varphi \subseteq [\![\mathbb{S}']\!]$, so it suffices to prove that $[\![\mathbb{S}']\!] \subseteq [\![\mathbb{S}]\!]$. Indeed, take $\tau \in [\![\mathbb{S}']\!]$, $\sigma_l \in \mathsf{LStore}, v \in X$ arbitrary and assume that $h_0 \in [\![p_h * p_a(v) * \text{True}_{\mathcal{A}}]\!]_\lambda$, where $p_h = \mathcal{W}[\![P_h \star P]\!]^{\sigma_0 \circ \sigma_l}_{\mathcal{A}}$, $p_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star x \in X]\!]^{\sigma_l}_{\mathcal{A}}$, $p'_h = \mathcal{W}[\![P_h]\!]^{\sigma_0 \circ \sigma_l}_{\mathcal{A}}$ and $p'_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star P \star x \in X]\!]^{\sigma_l}_{\mathcal{A}}$. From the well-formedness of the premise, find that $pv(P) = \emptyset$, so $p'_a(v)$ is well-defined, and the definitions of world semantics imply that $h_0 \in [\![p'_h * p'_a(v) * \text{True}_{\mathcal{A}}]\!]_\lambda$. As $\tau \in [\![\mathbb{S}']\!]$, find $\exists S'.\ \tau \models_{\mathbb{S}'} \sigma_l, (p'_h, p'_a, v), S'$. Lemma D.1 tells us $\exists p''_h, S.\ \tau \models_{\mathbb{S}} \sigma_l, (p''_h, p_a, v), S$ where $p_h * p_a(v) = p'_h * p'_a(v)$ - conclude that $p_h = p''_h$ and finally that $\tau \in [\![\mathbb{S}]\!]$. □

D.4 Atomic

Theorem D.9 (Soundness of OPEN REGION)**.** Let $\Phi \in \mathsf{FSpec}, \mathbb{C} \in \mathsf{Cmd}$ and $\mathbb{S} \in \mathsf{Spec}$. The following holds:

$$\textbf{if } \vdash_\Phi \mathbb{C} : \mathbb{S} \text{ by application of OPEN REGION}$$
$$\textbf{then } \models_\Phi \mathbb{C} : \mathbb{S}.$$

Proof. Assume $\vdash_\Phi \mathbb{C} : \mathbb{S}$ by an application of Open Region. Then for

$$\mathbb{S} = \forall x \in X.\langle\ P_h \mid P_a(x) \star \mathbf{t}^\lambda_r(x) \star \lceil G(x) \rceil_r\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid \exists z.\ Q_a(x,y,z) \star \mathbf{t}^\lambda_r(z) \star R(x,z)\ \rangle_{\lambda+1,\mathcal{A}}$$
$$\mathbb{S}' = \forall x \in X.\langle\ P_h \mid P_a(x) \star I(\mathbf{t}^\lambda_r(x)) \star \lceil G(x) \rceil_r\ \rangle\ \mathbb{C}\ \exists y.\langle\ Q_h(x,y) \mid \exists z.\ Q_a(x,y,z) \star I(\mathbf{t}^\lambda_r(z)) \star R(x,z)\ \rangle_{\lambda,\mathcal{A}}$$

From the premises of OPEN REGION,

$$r \in dom(\mathcal{A}) \implies R = id \tag{D.8}$$
$$\forall x \in X.\ (x,z) \in \{(x,z) | x \in X \wedge R(x,z) \wedge (x,z) \in \mathcal{T}_{\mathbf{t}}(G(x))^*\} \tag{D.9}$$
$$\vdash_\Phi \mathbb{C} : \mathbb{S}' \tag{D.10}$$

Henceforth, let

$$p_a = \lambda x.\ \begin{cases} \mathcal{W}[\![P_a(x) \star \mathbf{t}^\lambda_r(x) \star \lceil G(x) \rceil_r \star x \in X]\!]^{\sigma_l}_{\mathcal{A}} & x \in \mathsf{AVal} \\ \mathtt{Emp}_{\mathcal{A}} & \text{otherwise} \end{cases}$$
$$p'_a = \lambda x.\ \begin{cases} \mathcal{W}[\![P_a(x) \star I(\mathbf{t}^\lambda_r(x)) \star \lceil G(x) \rceil_r \star x \in X]\!]^{\sigma_l}_{\mathcal{A}} & x \in \mathsf{AVal} \\ \mathtt{Emp}_{\mathcal{A}} & \text{otherwise} \end{cases}$$

From the inductive hypothesis, we have $\models_\Phi \mathbb{C} : \mathbb{S}'$ so it suffices to prove $[\![\mathbb{S}']\!] \subseteq [\![\mathbb{S}]\!]$. Let

$$p_h = \mathcal{W}[\![P_h]\!]_{\mathcal{A}}^{\sigma_0 \circ \sigma_l}$$
$$p_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star \mathbf{t}_r^\lambda(x) \star \lceil G(x) \rceil_r \star x \in X]\!]_{\mathcal{A}}^{\sigma_l}$$
$$p'_a = \lambda x.\ \mathcal{W}[\![P_a(x) \star I(\mathbf{t}_r^\lambda(x)) \star \lceil G(x) \rceil_r \star x \in X]\!]_{\mathcal{A}}^{\sigma_l}$$

Take some $(\sigma_0, h_0, \mathbb{C})\tau \in [\![\mathbb{S}']\!]$, $\sigma_l \in \mathsf{LStore}, v \in X$ arbitrary and assume $h_0 \in [\![p_h * p_a(v) * \mathrm{True}_{\mathcal{A}}]\!]_{\lambda+1}$. Find some $w_h \in p_h, w_a \in p_a(v), w_f$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \rfloor_{\lambda+1}$. From world composition, it must be that $w_h = (\_, \rho, \_, \_)$, $w_a = (\_, \rho, \_, \_)$, $w_f = (\_, \rho, \_, \_)$ for some ρ such that $\rho(r) = (\mathbf{t}, \lambda, v)$. It is clear from the definition of reification that for $\mathrm{closed}_\lambda^{\lambda+1}(\rho) = \{r, r_1, ..., r_n\}$ and $\rho(r_i) = (\mathbf{t}_i, \lambda, a_i)$, there exists some $w_i \in \mathcal{I}_{\mathbf{t}_i}[\![r_i, \lambda, a_i]\!]$ and $w_i \in \mathcal{I}_{\mathbf{t}}[\![r, \lambda, v]\!]$ such that $h_0 \in \lfloor w_h \bullet w_a \bullet w_f \bullet w_r \bullet w_1 \bullet ... \bullet w_n \rfloor_\lambda$. $w_a \bullet w_r \in p'_a(v)$ so $h_0 \in [\![p_h * p'_a(v) * \mathrm{True}_{\mathcal{A}}]\!]$. Therefore from $\tau \in [\![\mathbb{S}']\!]$ conclude $\exists S'.\ \tau \models_{\mathbb{S}'} \sigma_l, (p_h, p'_a, v), S'$. Lemma C.5 tells us that $\exists S.\ \tau \models_{\mathbb{S}} \sigma_l, (p_h, p_a, v), S$ so $\tau \in [\![\mathbb{S}]\!]$. □