INDIANA UNIVERSITY BLOOMINGTON

Elizabeth Dietrich

A beginner's guide to Iris, Coq and separation logic

Bachelor's Thesis

Supervisor(s): Dennis Shasha

2021

A beginner's guide to Iris, Coq and separation logic

Abstract: Creating safe concurrent algorithms is challenging and error-prone. For this reason, a formal verification framework is necessary especially when those concurrent algorithms are used in safety-critical systems. The goal of this guide is to provide resources for beginners to get started in their journey of formal verification using the powerful tool Iris. The difference between this guide and many others is that it provides (i) an in-depth explanation of examples and tactics, (ii) an explicit discussion of separation logic, and (iii) a thorough coverage of Iris and Coq. References to other guides and to papers are included throughout to provide readers with resources through which to continue their learning.

Contents

1 Introduction

Iris is a higher-order separation logic framework for reasoning about concurrent programs. This guide explains how to use Iris in formal verification. We refer to formal verification as a way to use mathematical techniques to prove, in a rigorous and machine-checkable manner, the absence of bugs and the conformity of a system to its intended specification.

Unfortunately, Iris is difficult to learn. This guide attempts to teach you how to approach an Iris proof on your own. We assume no prior knowledge of Hoare logic or Coq and provide a section on separation logic to help build your intuition. This guide should serve as a starting point to your journey in learning Iris and provides links to resources throughout the sections for you to delve deeper into formal verification.

1.1 Case Studies

In Section 8 we implement a counter that increments by one in HeapLang, a simple default language for Iris. HeapLang is a functional language with mutable references that allows us to write directly from Coq (Chajed, 2020). We will delve more into the specifics of HeapLang in Section 6.1. We break the proof down into its different components that will allow us to gain intuition about Iris and separation logic. We show two different implementations of this counter. The first implementation is a simple program that will be explained in-depth in Section 5.7 and the second implementation will be a more advanced parallel increment example. This parallel increment example is fully explained in the example files of the Iris Project lecture notes. Therefore, we will not re-explain the example in its entirety here, but we will highlight its invariant.

The case study example will evolve into a more advanced bank example which allows the transfer of money between two bank accounts. The balance of these accounts will be represented as mathematical integers and their sum will be 0. Before we allow transfer from one account to the other we will need to ensure that the accounts have a sum of 0. The bank example will be fully explained in Section 9.

A more general bank example is taken from the guide: "A Brief Introduction to Iris". For readers looking to further their expertise after working through this guide, we suggest that as a starting point.

These examples are meant to be easy enough for you to try to re-implement on your own after working through this guide. For more examples, many of which are advanced, refer to the Iris Project lecture notes.

1.2 Summary and Outline

This guide explains the verification tools Iris and Coq, along with the basics of separation logic necessary to understand how to use these tools.

- Section 2 covers how to install all of the software we use throughout this guide including opam, Coq, and Iris.
- Section 3 introduces the Coq proof assistant. We present a simple proof to introduce the basics of proof mode, the type of reasoning Coq utilizes, and some of the main tactics available.
- In order to fully understand what is happening in Iris, we introduce the basics of separation logic in Section 4.
- Before we dive into the uses of Iris, we take a brief look at the base logic of Iris in Section 5. Here we further explain our simple increment example using proof rules and Hoare logic.
- Section 6 introduces the notation and some basic tactics of Iris. It presents how each tactic works and provides some examples on how a tactic might affect the state of your proof goal.
- Section 7 covers what is required to build your own proof in Iris from the ground-up.

This guide aims to be accessible to readers who have no prior background in formal verification. We hope that this tutorial on the Iris and Coq verification tools will allow you to start your journey into formal verification using these tools for proof mechanization. Further resources are provided throughout this guide to provide you with some possible next steps in learning more about Iris and Coq.

2 Installing the necessary software

2.1 Installing Opam

Before you can install Iris and Coq, you must ensure you have opam (version 2.0.0 or newer) installed on your device. Opam is the package manager for OCaml, the programming language in which Coq is implemented, that supports multiple simulataneous compiler installations and flexible package constraints. In order to install opam using Homebrew use command: `brew install opam`. Once opam is installed, it must be intialized before you can use it. We do this by the following commands in order: `opam init`, `eval $(opam env)`. `opam init` will prompt you to allow opam to set up intialization scripts which is generally fine to accept. If you do not accept, every time a new shell is opened, you will have to type the `eval $(opam env)` command to update environment variables. For further help with installing opam, including different installation avenues, refer to the opam documentation.

2.2 Installing Coq

While we briefly mention how to install opam above, the Coq platform provides interactive scripts that allow installing Coq and its packages through opam without having to learn anything about opam. You can use the command, `opam install coq` to install Coq on your device. Depending on your operating system, installing Coq using opam may require you to first install system packages. You can check the guide to installing Coq to see your personal device needs, here you will also find how to install the CoqIDE and other Coq packages.

2.3 Installing Iris

In this guide we refer to Iris implemented and verified in the Coq proof assistant. Therefore, in terms of installation we will present the Coq development of the Iris Project, a general proof mode for carrying out separation logic proofs in Coq. A more in-depth guide can be found here. To use Iris in your own proofs, you should install Iris via opam (2.0.0 or newer). In order to obtain the latest stable release, you have to add the Coq opam repository as follows:

```
opam repo add coq-released https://coq.inria.fr/opam/released
```

Now you can install Iris:

- `opam install coq-iris` will install the libraries making up the Iris logic, but leave it up to you to instantiate the program_logic.language interface to define a programming language for Iris to reason about.

- `opam install coq-iris-heap-lang` will additionally install HeapLang, the default language used by various Iris projects.

This guide assumes both commands are called by the user. To fetch updates later, run `opam update` and `opam upgrade`.

Once you have Iris installed, you can set up your editor to properly input and output the unicode characters used throughout Iris. While this guide will not provide instructions, please refer to this in-depth guide for further instructions. However, we do include an Appendix to serve as a guide for inputting Unicode symbols that applies to Visual Studio Code with the 'Generic Input Method' extension installed and configured as described in the guide previously mentioned.

3 What is Coq

Coq is a formal proof management system. It provides a rich dependently-typed framework to formalize mathematics and programming language metatheory. In order to make proofs feasible to construct, Coq supports a range of built-in *tactics* which are engineered primarily to support *backward reasoning*. When programming in Coq, you can use these tactics to manipulate the proof state interactively, applying axioms or lemmas to break the goal into subgoals until all subgoals have been solved. This allows Coq to infer many details about how to instantiate lemmas by inspection of the current proof state, and as a result the Coq user can omit these details from their interactive proof scripts.

3.1 Basics of proof writing

Coq is an interactive proof assistant, which means that proofs can be constructed interactively through a dialog between the user and the assistant. The building blocks for this dialog are tactics that the user can use to represent steps in the proof of a theorem.

3.1.1 Proof Mode

Proof mode is used to prove theorems. It is the core mechanism of the dialog between the user and the proof assistant. Coq enters proof mode when you begin a proof, such as with the **Theorem** or **Lemma** command, and it exits proof mode when you complete a proof, such as with the **Qed** command. Tactics, which are available only in proof mode, incrementally transform incomplete proofs to eventually generate a complete proof and will be discussed more below in Section 3.1.1.2. When you run Coq interactively through the CoqIDE or coqtop, which will be shown later, Coq shows the current proof state (the incomplete proof) as you enter tactics.

3.1.1.1 Proof State

The proof state consists of one or more unproven goals. Each goal has a conclusion, which is the statement that needs to be proven, and a local context, which contains named hypotheses (which are propositions), variables, and local definitions that can be used in proving the conclusion. The proof may also use constants from the global environment such as definitions and already proven theorems. The term goal may refer to either an entire goal or to the conclusion of a goal, depending on the context.

The conclusion appears below a line and the local context appears above the line. The local context of a goal contains items specific to the goal as well as section-local variables and hypotheses defined in the current section. The latter are included in the initial proof state. Items in the local context are ordered; an item can refer only to items that appear before it. The global environment has definitions and proven theorems that are global in scope.

```
Lemma example: ∀ b: bool, b=true ∨ b=false.
Proof.
intros. destruct b. left. done. right. done.
Qed.
```

Figure 1. Proof that every Boolean is either true or false.

Let us look at the example lemma in Figure 1 that will prove every Boolean is either true or false. When you begin proving a lemma, the proof state shows the statement of the theorem below the line and often nothing in the local context, as seen in Figure 2.

```
(1/1)
∀ b : bool, b = true ∨ b = false
```

Figure 2. After defining the Boolean lemma, we use tactic **Proof** to start our proof. This gives us the proof state of the theorem we must now prove.

After applying the **intros** tactics, we see the hypothesis: $b = true \vee b = false$. This also introduces the term "b" into our local context. The name of the variable (b) appears before the colon, followed by the type (bool) that it represents which can be seen in Figure 3.

```
b : bool
```

```
(1/1)
b = true ∨ b = false
```

Figure 3. After using the tactic **intros** we need to prove $b = true \vee b = false$ and the term "b" was added to our local context.

"Variables" may refer specifically to local context items and "hypotheses" refers to items that are propositions. However, these terms are often used interchangeably. We use the knowledge that b is a bool by calling the tactic **destruct** in Figure 4 which will split the proof according to the two cases.

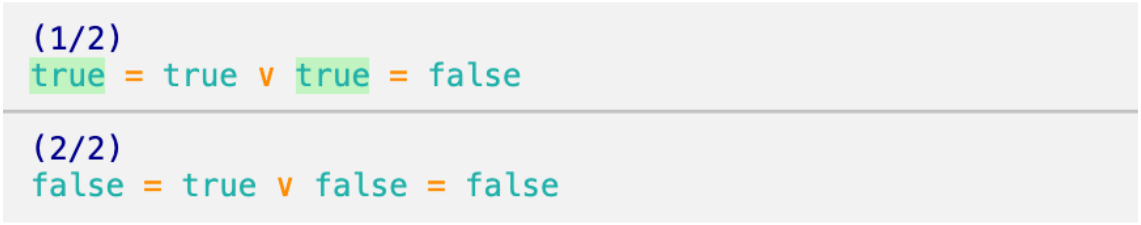

Figure 4. The tactic **destruct b** splits the proof according to the two cases and takes b from our local context to show all of the possibilities of its value. This is an example of a state with two subgoals.

In each case, it is easy to determine that both $true = true$ and $false = false$ is trivial. This means we can finish the proof off by trivially showing the left or right hand side of each case is true. In order to do so, we can use the tactics **left** and **right** for each respective side. These tactics will take the correct side of our logical disjunction. For example, **left** will result in the statement $true = true$ of our first subgoal. After we use these tactics, we can call the tactic **done** on both cases since this statement is trivially correct. We then finish our proof and exit proof mode through the **Qed** tactic. Coq's kernel verifies the correctness of proof terms when it exits proof mode by checking that the proof term is well-typed and that its type is the same as the theorem statement. The full proof involving these tactics can be seen in Figure 1. When we write "proof" we refer to a proof script that consists of the tactics that are applied to prove a theorem.

3.1.1.2 Tactics

Tactics are available only in proof mode. They specify how to transform the proof state of an incomplete proof to eventually generate a complete proof. Coq and its tactics use backward reasoning. In backward reasoning, the proof begins with the theorem statement as the goal which is then gradually transformed until every subgoal generated along the way has been proven. For example, take the proof of $A \wedge B$. This proof would begin the formula $A \wedge B$ as the goal. This can then be transformed into two subgoals A and B, followed by the proofs of A and B.

A tactic may fully prove a goal, in which case the goal is removed from the proof state. However, more commonly, a tactic replaces a goal with one or more subgoals. Most tactics require specific elements or preconditions to reduce a goal and will display an error message if the tactic cannot be applied to the goal. Tactics are applied to the current goal by default.

While a few tactics were discussed in the example above, we will not delve deeper

into the specifics of Coq tactics in this guide, as we focus on using Iris tactics as found in Section 6.2. An in-depth guide to Coq tactics can be found in the reference manual of Coq.

4 Separation Logic

In this user guide, we use *separation logic* to specify and verify concurrent data structures. We based this section largely off of the text of (Krishna et al., (In Press)). Separation logic (SL), an extension of Hoare logic, is used for reasoning about programs that access and mutate data held in computer memory. SL is based on the separating conjunction $P * Q$, which asserts that *P* and *Q* hold for separate portions of memory and allows one to modularly describe states of a program through proof rules that rely on separation (O'Hearn, 2012). Each sentence in this language is called a *formula*, and describes a set of states. We say a state *satisfies* a formula when the state is described by the formula. The set of proof rules that SL gives us can be used to prove that states of interest (such as the set of resulting states after a program executes) satisfy a particular formula. SL can be used in many ways including automatic program-proof tools or abstract interpreters.

Here, we will look at SL's relationship to *Iris*, a framework for higher-order concurrent separation logic, implemented in the Coq proof assistant. While this guide will touch only on the basics of SL, the core Iris logic is very abstract. We refer readers to the paper Iris from the ground up for a more in-depth look into this logic.

4.1 Basics

In this section we will introduce the basic constructs of SL before we consider the formal definitions and formulas that we will need to understand Iris.

4.1.1 "Points-To" Connective

Separation logic is a *resource* logic; not only do propositions denote facts about the state of the program, but also *ownership* of resources. In this section, we fix the notion of a resource to be a heap, a piece of global memory represented as a finite partial mapping from memory locations to the values stored there. The *points-to* connective, $p \mapsto q$, asserts that the program state contains a heap cell at address p that contains value q. If we are verifying an expression *e* that has $p \mapsto q$ in its precondition, then we can assume not only that the location *p* currently points to *q*, but also that the *right* to update *p* is *owned* by *e*. Ownership, in this case, means that when verifying *e* we would not need to consider the possibility that another piece of code in the program might interfere with *e* by updating *p* during *e*'s execution (Jung et al., 2018). We will further elaborate on this on the next page.

4.1.2 Separating Conjunction

The separating conjunction, $*$ (also known as *star*), allows us to state properties in disjoint parts of the memory. Unlike the standard conjunction $\wedge$, the separating conjunction

conjoins two formulas that describe *disjoint* portions of a program state. For instance, $x \mapsto \_ \wedge y \mapsto \_$ asserts that x is a heap cell and y is a heap cell (but they could be the same heap cell), while $x \mapsto \_ * y \mapsto \_$ asserts that x is a heap cell *and separately* y is a heap cell.

Formally, a state σ satisfies $P * Q$ if it can be broken up into two disjoint states $\sigma = \sigma_1 \circledcirc \sigma_2$ such that σ_1 satisfies P and σ_2 satisfies Q ($\circledcirc$ is a partial operator on states that is effectively disjoint union). When P and Q are formulas denoting heaps, they talk about disjoint regions of the heap, i.e. they do not have any heap addresses in common. In particular, this means that $x \mapsto \_ * y \mapsto \_$ implies that $x \neq y$, while the formula $x \mapsto y * x \mapsto z$ is unsatisfiable as it is not possible to split a heap into two disjoint portions both of which contain the same address x.

In terms of basic concurrent separation logic, propositions mean almost the same as in traditional separation logic; however, we denote ownership by whichever *thread* (execution context) is running the code in question. In the context of the separating conjunction, this means that if a thread *t* can assert $p \mapsto q$, then t knows that no other thread can read or write *p* concurrently, so it can completely ignore the other threads and just reason about *p* as if it were operating in a sequential setting.

4.1.3 Separating Implication

The separating implication connective, $-\!\!*$ (also known as a *magic wand*), asserts that whenever a fresh heap satisfies a property, its composition with the current heap satisfies another property. This is most useful when a piece of code mutates memory locally, but we want to state some property of the entire memory (Demri and Deters, 2015).

Formally, a state σ satisfies $P -\!\!* Q$ if for every state σ_1 that is disjoint from σ and that satisfies P, the combined state $\sigma \circledcirc \sigma_1$ satisfies Q. The best way to understand $-\!\!*$ is to think of $*$ and $-\!\!*$ as separation logic analogues of $\wedge$ and $\Longrightarrow$ respectively from first-order logic. For instance, $P * Q$ means you have both P *and* Q (and that they are disjoint). Similarly, $P -\!\!* Q$ describes a state such that if you conjoin it with a state satisfying P, then you get a state satisfying Q. This property, $P * (P -\!\!* Q) \vdash Q$, is the SL analogue of *modus ponens* in first-order logic $P \wedge (P \Longrightarrow Q) \vdash Q$.

5 Iris Base Logic

Iris is a higher-order concurrent separation logic framework. It defines a program logic for a generic language specified in terms of its expressions, values, and operational semantics; it also provides a weakest-precondition based program logic. Weakest-precondition based program logic stems from Dijkstra's weakest-precondition calculus and is a technique for proving properties of imperative programs.

In this guide we will not provide the grammar and propositions of separation logic because we will show the in-depth discussion of these topics specifically for Iris in this section. We will also refer to Hoare logic specifically in relation to separation logic and Iris. However, it is important to note that Hoare logic is a more general concept and can be used without separation logic for more information refer to Hoare's original paper.

5.1 Propositions

Iris propositions describe the *resources* owned by a thread as was described above in Section 4.1.1. We will focus on the simple case where propositions describe concrete program states in the form of subsets of the heap. We defer the discussion of advanced resources to more advanced readings[1].

$$
\begin{aligned}
P, Q, R := {} & TRUE \mid FALSE \mid P \land Q \mid P \lor Q \mid P \Rightarrow Q && (1)\\
\mid {} & \exists x.\ P \mid \forall x.\ P && (2)\\
\mid {} & x \mapsto v \mid P * Q \mid P \mathrel{-\!\!*} Q && (3)\\
\mid {} & \boxed{P}^{N} \mid \triangleright P \mid \ulcorner a \urcorner^{\gamma} && (4)\\
\mid {} & \{P\}e\{v.Q\} \mid \langle x.P \rangle e \langle v.Q \rangle \mid && (5)
\end{aligned}
$$

Figure 5. The grammar of Iris propositions used in this user guide.

The grammar of the subset of Iris propositions that we use throughout the user guide is shown in Figure 5.1 and includes the following constructs:

- The first line consists of standard propositional constructs: the propositions $TRUE$ and $FALSE$, conjunction, disjunction, and implication.
- The second line introduces quantification. What makes Iris a *higher-order* logic is that universal and existential quantifiers can range over any type. Hence, x can range over any type, including that of propositions and (higher-order) predicates.

[1] Lecture Notes on Iris: Higher-Order Concurrent Separation Logic provides an in-depth look into many Iris features, including intuition for Iris propositions

- We have already seen the points-to proposition, separating conjunction, and separating implication on the third line.
- The fourth line contains the invariant proposition $\boxed{P}^N$, explained in 5.3, the later modality $\triangleright P$, explained in 6.2.9, and the ghost state proposition $\boxed{a}^\gamma$, explained in 5.4.
- Finally, the last line contains the Hoare and atomic triples which will be explained in 5.6.

5.2 Resource Algebra

What is a resource? The Iris base logic does not answer this question by fixing a particular set of resources. Instead, the set of resources is kept general and is up to the user of the logic to make a suitable choice. All the logic demands is that the set of resources forms a unital resource algebra (RA)[2] (Krebbers et al., 2017). We see the formal definition of an RA in Figure 6.

A *resource algebra* is a tuple $(M, \mathcal{V} : M \to Prop, (\cdot) : M \times M \to M, \epsilon \in M)$ satisfying:

$$\begin{aligned}
\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) &\qquad RA - ASSOC \\
\forall a, b.\ a \cdot b = b \cdot a &\qquad RA - COMM \\
\forall a.\ \epsilon \cdot a = a &\qquad RA - ID \\
\mathcal{V}(\epsilon) &\qquad RA - VALID - ID \\
\forall a, b.\ \mathcal{V}(a \cdot b) \Rightarrow \mathcal{V}(a) &\qquad RA - VALID - OP
\end{aligned}$$

Figure 6. The definition of a (unital) resource algebra (RA).

Formally, a resource algebra (RA) consists of a set M, a validity predicate $\mathcal{V}$(-), and a binary operation$(\cdot) : M \times M \to M$ that satisfy the axioms in Figure 6 (Prop is the type of propositions of the meta-logic (e.g., Coq)). RAs are a generalization of the partial commutative monoid (PCM) algebra commonly used by separation logics. In Figure 6, we see the resources owned by different threads can be composed using the $\cdot$ operator, composition of ownership is associative and commutative (reflecting the associative and commutative semantics of parallel composition), and combinations of ownership that do not make sense are ruled out by validity (when multiple threads claim to have

[2]Iris actually uses *cameras* as the structure underlying resources, but since we do not use higher-order resources or states which can embed propositions in this guide, we restrict our attention to resource algebras, a stronger, but simpler, structure.

ownership of an exclusive resource), specifically a subset, $\mathcal{V}$, of *valid* elements. This take on partiality will be necessary when we define higher-order ghost states that are needed for modeling invariants which we will discuss in Section 5.4 (this is because the composition of all resources at a ghost location must always be valid). This is merely a brief overview of RAs to introduce you to their nature, we encourage readers to delve more into the specifics after grasping the basics first.

5.3 Invariants

We need to reason compositionally about shared state. Threads take turns interacting with the shared state (reading/writing, etc.). However, how do we verify one thread at a time even though other threads may interfere with, or mutate, the shared state in between each step of computation in the thread we are verifying? Concurrent program logic accounts for such interference via *invariants*. An invariant is a property that holds on some piece of shared state at all times: each thread accessing the state may assume the invariant holds before each step of its computation, but it must also ensure that it continues to hold after each step (Ralf et al., 2015).

In Iris, an invariant is a proposition of the form $\boxed{P}^N$ where P is an arbitrary Iris proposition. Intuitively, an invariant is a property that, once established, will remain true forever. Therefore, it is a duplicable resource and can be freely shared with any thread. In order to ensure an invariant remains true once it has been established, Iris's proof rules for invariants impose restrictions on how the resources contained in the invariant can be accessed and manipulated. A thread can *open* an invariant $\boxed{P}^N$ to gain ownership of the contained resources P. These resources can then be used in the proof of a single atomic step of the thread's execution. After the thread has performed an atomic step with an open invariant, the invariant must be closed, which amounts to proving that P has been reestablished. Otherwise, the proof will not succeed.

In our denotation of an invariant, $\boxed{P}^N$, the N refers to the namespace of the invariant. A namespace is the part of the mechanism used in Iris to keep track of invariants that are currently open and need to be closed before the next atomic step. This ensures we avoid issues of re-entrancy in case of nested invariants which would lead to logical inconsistencies.

In Section 9 we explain a "parallel increment" example. In this example we define the body of an invariant and explain how we open and close that invariant. Through this process we will see the main tactics used for this type of procedure.

5.4 Modalities

Modalities are expressions that qualify assertions about the truth of statements. Iris extends traditional separation logic with a handful of modalities - such as the persistence

modality ($\Box$), the step-indexed "later" modality ($\triangleright$), and the (basic) update modality ($\Rrightarrow P$). Iris proof mode provides tactical support for reasoning conveniently about these modalities, some of which are introduced in Section 6.2.9.

- Persistence modality : denoted by $\Box P$, asserts that *P* holds without asserting any exclusive ownership. An assumption, $\Box P$, can be used arbitrarily often, it cannot be "used up". For example, $P \twoheadrightarrow Q$ is a linear implication that can be applied only once. However, $\Box(P \twoheadrightarrow Q)$ can be applied arbitrarily often (Jung et al., 2018). Note that this can be used to encode Hoare triples (explained in Section 5.6). An assertion *P* is persistent if proofs of *P* can never assert exclusive ownership, this means that these propositions can be freely duplicated, the usual restriction of resources being usable only once does not apply to them.

- Later modality : denoted by $\triangleright P$, asserts that *P* holds at the next step-index or "one step later". Iris assertions can serve as an invariant, so one can define invariants that refer to other invariants or even themselves. This introduces a potential circularity. The notion of steps using a later modality introduces step-indexing which allows us to break this circularity. To ensure that invariants are well defined, the shared resource backing $\boxed{P}^N$ need satisfy only $\triangleright P$. Therefore, opening an invariant grants ownership of the shared resource one step later and closing the invariant reestablishes the shared resource one step later (Ralf et al., 2015). Thus we see the later modality is needed to ensure soundness in complex cases when Hoare triples or invariants themselves are stored inside invariants. If we do not store Hoare triples or invariants inside invariants, we can generally ignore the later modality, which we will do throughout this guide (Jung et al., 2018).

- (Basic) Update modality : denoted by $\Rrightarrow P$, allows you to talk about what you could own after performing an update to what you do own (Krebbers et al., 2017). It reflects frame-preserving updates into the logic, in the sense that $\Rrightarrow P$ asserts ownership of resources that can be updated to resources satisfying *P*. A frame-preserving update is an update to a resource. If we think of frames as being the resources owned be other threads, then a frame-preserving update is guaranteed not to invalidate the resources of concurrently-running threads. There is another modality, the fancy update modality or view shift, that has a similar, but more advanced purpose which will not be covered in this guide but is covered more in The Essence of Higher-Order Concurrent Separation Logic paper.

5.5 Ghost State

Ghost State, originally called auxiliary variables, is a formal technique where the prover adds state (variables or resources) to a program that captures knowledge about the history

of a computation not present in the state of the original program in order to verify it. Ghost state was originally proposed as a way to abstractly characterize some knowledge about the history of a computation that is essential to verifying it. It is used for modularly describing a thread's knowledge about some shared state as well as the rights it has to modify it. Ghost state is a purely logical construct, introduced solely for the purpose of verification, and is not a part of the program syntax. Since in Iris, ghost state is purely logical and ghost commands are represented as proof rules, we know that the added ghost state and the ghost commands that modify it, have no effect on the run-time behavior of the program. The proof of this augmented program can be transformed into a proof of the program with all ghost states removed (i.e., the original program)[3].

5.6 Hoare Triples and Atomic Triples

Hoare Triples are assertions of the form $\{P\}e\{v.Q\}$, where P and Q are formulas. $\{P\}e\{v.Q\}$ is true if for every state σ that satisfies P we have that (1) the program e does not reach an error state when run from σ (for example, by trying to read unallocated memory), and (2) that if e terminates then it returns some value v and the new state is some σ' that satisfies Q. We call P the *precondition* and Q the *postcondition* of e, and we write $\{P\}e\{Q\}$ in the case where Q does not mention the return value v. Let us look at an example of a Hoare Triple:

$$\{X = 1\}X := X + 2\{X = 3\}$$

Here P is the condition that the value of X is 1, Q is the condition that the value of X is 3, and e is the assignment command $X := X + 2$ i.e. 'X becomes X+2'. We know this is a valid Hoare Triple as the command X := X + 2 transforms a state in which X = 1 to a state in which X = 3 (Gordon, 2014).

Recall the mention of weakest-precondition from the beginning of this section and let us break down what this means using Hoare Logic. Given a precondition P, a command e, and a postcondition Q such that $\{P\}e\{Q\}$, we want to find the unique P that is the weakest precondition for e and Q. This means that if e is a command and Q is an assertion about states, then the weakest precondition for e with respect to Q is an assertion that is true for precisely those initial states from which e must terminate and executing e must produce a state that satisfies Q. Let us see this in an example where e is $X := X + 1$ and Q is $(X > 0)$. One valid precondition would be $(X > 0)$ as we see the following Hoare statement is valid:

$$\{X > 0\}X := X + 1\{X > 0\}$$

Another valid precondition would be $(X > -1)$ as the following Hoare statement is also valid:

$$\{X > -1\}X := X + 1\{X > 0\}$$

[3]Iris does not distinguish between real and ghost states, all states are considered to be ghost states.

We can see that $(X > -1)$ is weaker than $(X > 0)$ since $(X > 0) \Rightarrow (X > -1)$ and in fact, $(X > -1)$ is our weakest precondition (Goré, 2016).

Logical atomicity is a useful concept that is used to reason about concurrent structures. It is used to specify programs that execute in multiple atomic steps but whose effect appears to take place in a single point in time. This concept will be useful for specifying concurrent structures in a way that can be used to verify concurrent client programs.

We can specify the concurrent behavior of programs using *atomic triples*. An atomic triple $\langle x.P \rangle e \langle v.Q \rangle$ is made up of the pre-condition P, return value v, post-condition Q, and a program e. Such a triple means that e, if it terminates, despite executing in potentially many atomic steps, appears to operate atomically on the shared state and transforms it from a state satisfying P to one satisfying Q. This means there is a single physical step during the execution of e when the shared state is transformed from P to Q.

For the sake of this guide, we briefly introduce the concept of atomic triples before diving into proof rules and propositions. We will introduce proof rules for Hoare and Atomic Triples in Section 5.7 so we can use them to reason about programs and examples in this guide.

5.7 Proof Rules

Before we can prove any specifications in separation logic, we must introduce some proof rules for Hoare Triples (Figure 2.1). A proof rule, or inference rule, consists of two parts separated by a horizontal line: the part above the line contains one or more premises, and the part below the line contains the conclusion. A rule with no premises is called an *axiom*, and in this case we omit the horizontal line. Some rules are bi-directional: the premise implies the conclusion, but the conclusion also implies the premise. These rules are denoted with a double horizontal line. We will explain some of the proof rules in Figure 7 as we discuss an example below. For more information on the proof rules, reference the Lecture Notes on Iris: Higher-Order Concurrent Separation Logic.

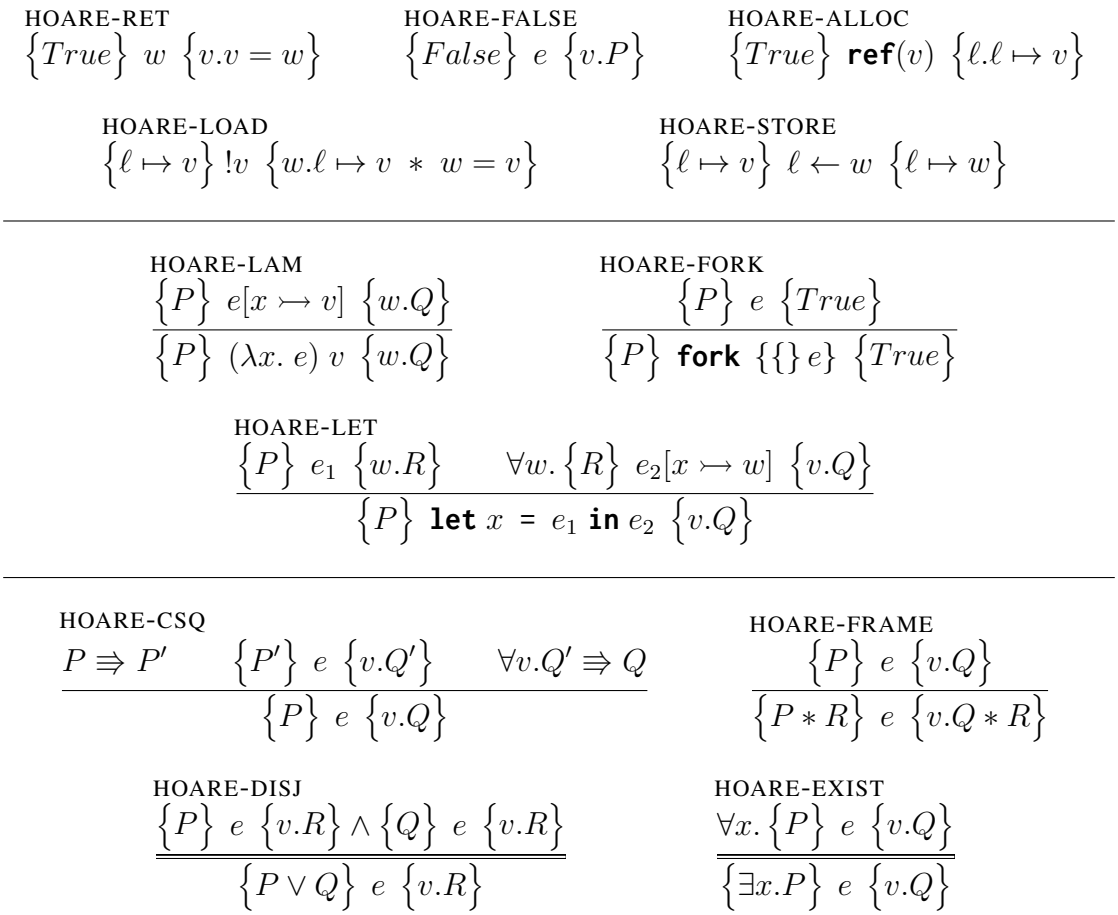

Figure 7. Some of the proof rules for establishing Hoare triples. We write $e[x \rightarrowtail v]$ to denote the expression e after substituting all occurrences of the variable x with the term v.

Recall our discussion of backward reasoning, we can apply this process to solving a proof tree. Like mentioned before, we will start with a specification that we want to prove, find an inference rule that matches the structure of the goal, and apply it. This replaces the goal with the premises of the rule.

Let us take a counter example to better understand how we prove a program in this style. Take the following program expression which reads the value stored at heap location x into a variable ($v\ = !x$) and then writes $v + 1$ back into the location:

$$e_{inc} := \mathtt{let}\ v\ =\ !x\ \mathtt{in}\ x \leftarrow (v + 1)$$

Essentially, this means e_{inc} is a program that increments the value stored at the heap location x. Before we can prove anything, we need to use Hoare Triples to write the

specifications of our program e_{inc} as follows:

$$\left\{x \mapsto n\right\}\ e_{inc}\ \left\{x \mapsto n+1\right\}$$

We see that here the precondition uses a points-to predicate $x \mapsto n$ to assert the program state contains a heap cell at address x containing value n and the postcondition uses a points-to predicate to assert the program state contains a heap cell at address x containing value $n+1$. In simple words, this Hoare Triple tells us that if the program e_{inc} is run on a state that contains a heap cell at address x with value n, then it results in a state where the cell x contains $n+1$.

We can now prove that e_{inc} meets this specification using our proof rules as follows:

$$\dfrac{\dfrac{hoare-load}{\left\{x \mapsto n\right\}\ !x\ \left\{v.\, x \mapsto n\ *\ v = n\right\}} \qquad \dfrac{\dfrac{\dfrac{}{\left\{x \mapsto n\right\}\ x \leftarrow (v+1)\ \left\{x \mapsto v+1\right\}}\ hoare-store}{\left\{x \mapsto n\ *\ v = n\right\}\ x \leftarrow (v+1)\ \left\{x \mapsto v+1\ *\ v = n\right\}}\ hoare-frame}{\forall v.\left\{x \mapsto n\ *\ v = n\right\}\ x \leftarrow (v+1)\ \left\{x \mapsto n+1\right\}}\ hoare-csq}{\left\{x \mapsto n\right\}\ \texttt{let}\ v\ \texttt{=}\ !x\ \texttt{in}\ x \leftarrow (v+1)\ \left\{x \mapsto n+1\right\}}\ hoare-let$$

When explaining a proof using backward reasoning, we use a proof tree. We read *proof trees* starting from the goal at the bottom or the root of the tree and work our way up. Each horizontal line represents the use of an inference rule to transform the current goal into one or more goals. As we apply more inference rules, our goal hopefully becomes simpler.

We will briefly explain our proof tree for e_{inc} above, although certain details will be emitted for readability for beginners. In this proof tree, we see the first rule we use is $HOARE - LET$, which works on let-bindings and breaks up the proof. Here we see it splits the proof of the bound expression $!x$ (on the left), and the proof of the let-body $x \leftarrow (v+1)$ (on the right). It is important to note that proof rule $HOARE - LET$ requires the user to come up with an intermediate proposition R that describes the states of the program after evaluating the bound expression e_1 but before evaluating the let-body e_2. In this particular example since e_1 is a dereference, our only option is to use the rule $HOARE - LOAD$, allowing us to solve the left sub-tree. Now let us consider the right sub-tree. Here our only option is to use $HOARE - STORE$, but the current proof goal does not quite fit. Specifically, $HOARE - STORE$ will give us the postcondition $x \mapsto v+1$ while we need the postcondition $x \mapsto n+1$. In order to infer this, we need the fact that $v = n$, which is something that we know in the precondition of the current goal. If we can transfer the fact $v = n$ from the precondition to the postcondition of the program fragment $x \leftarrow (v+1)$, then we can use $HOARE - CSQ$ to complete the proof tree.

Since we can think of propositions in separation logic as resources, let us think about the predicate $x \mapsto n$ as a resource since it describes a heap cell or a part of the program

state. The *frame rule*, $HOARE - FRAME$, allows us to *frame* resources around a program fragment. This means we can carry resources that are not needed by the program fragment from its precondition to its postcondition. In the case of our proof tree, the program fragment is $x \leftarrow (v + 1)$, and as this does not modify v or n, we frame the resource $v = n$, allowing us to use $HOARE - STORE$ to finish the proof.

As can be seen from this example, the structure of the program expression will often determine the proof rule we must use, as well as any intermediate proposition that links the first goal to the next one.

6 Iris in Action

For the purpose of this guide, we refer to Iris implemented and verified in the Coq proof assistant. This section is dedicated to the different pieces of Iris to give insight into the highly modular framework and how to use Iris in the Coq proof assistant (Chajed, 2020). Earlier sections introduce Iris base logic and the core logic that all of Iris is built upon. Later, we will also present further examples.

6.1 HeapLang

HeapLang is the standard example language for Iris. While it is not the only language we can reason about with Iris, it is a reasonable language for simple examples. HeapLang is a lambda-calculus programming language with mutable state recursive functions, general higher-order references and fork-style concurrency (the fork construct can be used to spawn new threads). Integers, Booleans, units, heap locations, and (binary) sums and products are built-in primitives (Simon et al., 2020; Jung, 2021b). It has syntax for Hoare triples and tactics for relatively easy program proofs of weakest preconditions. Here we will explain the basics of using HeapLang in a Coq proof assistant; however, for a more thorough list of tactics, please refer to the HeapLang documentation. An explanation on how to install HeapLang will be addressed in Section 2.3.

6.1.1 Tactics

HeapLang comes with tactics that facilitate stepping through HeapLang programs as part of proving a weakest precondition. Note that here, # is used as a short-hand to turn a basic literal (an integer, a boolean literal, etc.) into a value. Since values coerce to expressions, # is used whenever a Coq value needs to be placed into a HeapLang term. Now let us look at some important tactics that take one or more pure program steps:

- **wp_pure** : performs one pure reduction step. Pure steps include beta reduction, projections, constructors, as well as unary and binary arithmetic operators.
- **wp_pures** : performs as many pure reductions steps as possible. This tactic will not reduce lambdas that are hidden behind a definition. If the computation reaches a value, the WP will be entirely removed and the postcondition becomes the new goal.
- **wp_bind pat** : apply the bind rule to "focus" the term matching pat. This is useful when accessing invariants, which is possible only when the WP in the goal is for a single, atomic operation. wp_bind can be used to bring the goal into the right shape.

- **wp_apply proof_mode_term** : apply a lemma whose conclusion is a WP, automatically applying wp_bind as needed.
- **wp_load** : reduces a load operation and automatically will find the points-to assertion in the spatial context.
- **wp_store** : reduces a store operation and automatically will find the points-to assertion in the spatial context.
- **wp_op** : reduces unary, binary and other arithmetic operators.

All of the tactics of HeapLang assume that our current goal is of the form "WP e @ E {{ Q }}" (where WP stands for "weakest precondition"). Let us take a look at one of our examples from Section 4 to see how some of these tactics work. Take the Hoare triple: $\{X = 1\}X := X + 2\{X = 3\}$. We won't go into the implementation specifics in this section, but we will show how these tactics work once we have our "WP" format. As we can see in Figure 8. Here we will be looking at our program "inc_x" which stands for the function, X := X + 2. Since we see a goal in this format, we know to use HeapLang tactics.

```
Definition wp : expr :=
λ: "l", let: "n" := !"l" in
        "l" <- "n" + #2.

Lemma wp_example (ℓ : loc) (n : Z):
{{{ ℓ ↦ #1 }}} wp #ℓ {{{RET #(); ℓ ↦ #3}}}.

Proof.
iIntros (Φ) "Hpt HΦ.
wp_pures. wp_load. wp_store. iModIntro.
iApply "HΦ". iFrame.
Qed.
```

```
(1/1)
"Hpt" : ℓ ↦ #1
"HΦ" : ▷ (ℓ ↦ #3 -∗ Φ #())
--------------------------------------∗
WP wp #ℓ {{ v, Φ v }}
```

Figure 8. Our current goal is in the format "WP e @ E {{ Q }}".

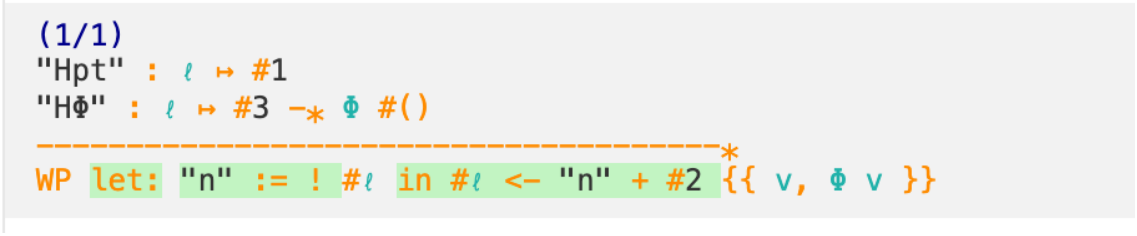

Figure 9. The tactic wp_pures reduces as many steps as possible.

For example, we will use the tactic **wp_pures**, one of the most common HeapLang tactics, to perform as many pure reduction steps as possible. This could potentially be the last HeapLang tactic we need. If our computation reaches a value, the WP will be entirely removed from the goal. However, we see in Figure 9 this is not the case.

Figure 10. The tactic wp_load will reduce the load operation and introduce #1 into our goal.

We know we must still use another HeapLang tactic before we can move on from this portion of the proof. We see in Figure 9 we see that in our *let* statement "n" := ! #ℓ. We want to reduce this load operation by using the tactic **wp_load**. This will take the "1" that we assumed in our precondition and introduce it into our goal. The last step we see after this is to somehow combine our 1 and 2 in the goal in order to get the 3 that we want. We can use the tactic **wp_store** to automatically find the points-to assertion in our goal that is "#ℓ <- "n" + #2" and combine it with "n" := #1. This will then get stored in our hypothesis "Hpt" as seen in Figure 11.

Figure 11. The tactic wp_store will automatically find the points-to assertion in our goal that is "#ℓ <- "n" + #2" and combine it with "n" := #1.

Since we no longer see a "WP" in our current goal we know that we have finished the weakest precondition portion of our proof and will no longer be using our HeapLang tactics. Although we show all of the code needed for this example below, Section 9 will provide further details into the implementation of similar proofs and a deep explanation into tactics required to start and finish such a proof.

6.2 Iris Proof Mode

The Iris Proof Mode (IPM) is an embedded domain specific language (DSL) that is used for interactive proofs in separation logic. It works for a large variety of separation logics. The IPM is actually mostly independent of Iris as the implementation uses only tactics and typeclasses. We will introduce many of the important tactics IPM provides here, however, for a more in-depth guide with more tactics, refer to the Iris tactic documentation. These tactics will be used throughout this guide to prove various examples.

The example we will utilize in this section to describe some of the tactics we introduce will be proving transitivity of x, y, and z. The whole code for this example is provided in Figure 12 but we will continue to break down individual tactics throughout this section. The tactics we do not provide an example for here will be explained in the case work at the end of this guide. There we will also explain the different parts of a proof and how to bring them all together. Below we introduce the full code for our transitivity example, but specific proof views and tactics will be explained as we progress through this section. Note that the tactics **rewrite**, **subst**, and **done** are Coq tactics so will not be explained here, for further information on their purpose, refer to Section 3.

```
Definition trans : expr :=
λ: "x",
  let: "y" := "x" in
  let: "z" := "y" in
  "x" = "y";;
  "y" = "z";;
  #().

Lemma transitivity (x y z: loc):
  {{{ ⌜x = y⌝ ∗ ⌜y = z⌝ ∗ ⌜x = x⌝}}} trans #x {{{RET #(); ⌜x = z⌝}}}.

Proof.
  iIntros (Φ) "Hxy HΦ".
  iRename "Hxy" into "Hx".
  iDestruct "Hx" as "(Hxy & Hyz & Hxx)".
  iClear "Hxx".
  iDestruct "Hxy" as %Hxy.
  iDestruct "Hyz" as %Hyz.
  wp_pures. iModIntro.
  iApply "HΦ".
  iAssert (⌜x = z⌝)%I as %yz.
  {
    iPureIntro. rewrite Hyz in Hxy. done.
  } subst z.
  iAssert (⌜x = y⌝)%I as %xy.
  {
    iPureIntro. done.
  } subst y. iFrame.
Qed.
```

Figure 12. Proving the transitivity of x, y, and z

6.2.1 Basic Terms

Before we look at our example and tactics, let us consider Figure 13 as an example proof view in IPM. The IPM embeds a separation logic context within the Coq goal. This means we have the Coq context above the solid line and the IPM context below the solid line. Inside the IPM context we have a persistent context (intuitionistic separation logic hypotheses) and a spatial context (spatial separation logic hypotheses). The persistent context is separated by - - - -□ and is composed of facts that are duplicable and do not go away when split, they can be used any number of times. The spatial context is separated by - - - -∗ and is composed of ordinary spatial premises. More explanation into these different contexts and the tactics that must be used for each one are explained in the sections below. To follow up this discussion, Section 9 contains in-depth examples of many of these tactics and how they might effect the context of a proof.

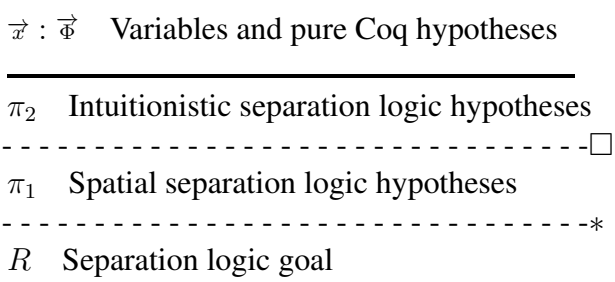

Figure 13. Example proof set-up in Iris Proof Mode that corresponds to proving: $\Pi_2 \twoheadrightarrow \Pi_1 \twoheadrightarrow R$

6.2.2 Introduction patterns (ipat)

Introduction patterns are used to perform introductions and eliminations of multiple connectives. This means we will either add a hypothesis or remove a hypothesis from our proof state. Some of the introduction patterns that the proof mode supports are:

- **H** : creates a hypothesis named H. In the transitivity example presented above, we do not create a hypothesis "H" but rather introduce hypotheses "Hxy" and "HΦ". We can later use these hypotheses to further break down our goal. Note that naming of these hypotheses is up to the user.

```
x, y, z :  loc
Φ :  val → iPropI Σ
```

```
(1/1)
"Hxy" : ⌜x = y⌝ ∗ ⌜y = z⌝ ∗ ⌜x = x⌝
"HΦ" : ▷ (⌜x = z⌝ -∗ Φ #())
--------------------------------------∗
WP trans #x {{ v, Φ v }}
```

Figure 14. The result of using the tactic iIntros (Φ) "Hxy HΦ" to introduce hypothesis "Hxy" and "HΦ into the spatial context of our proof state.

- **?** : creates an anonymous hypothesis (unnamed hypothesis).
- **_** : clears the hypothesis out of the proof view.
- **$** : frames the hypothesis in the goal of the proof view.

- [**ipat1 ipat2**] : (separating) conjunction elimination (separating conjunction explained in Section 4.1.1). In order to destruct conjunctions of the form $P \wedge Q$, either the proposition P or Q must be persistent *or* either ipat1 or ipat2 should be _ which allows the result of one of the conjuncts to be thrown away (hence it will disappear from the proof view).

Note that in case a branch of ipat starts with #, it causes the hypothesis to be moved to the intuitionistic context and if it starts with %, it causes the hypothesis to be moved to the pure Coq context.

6.2.3 Selection Patterns (selpat)

Selection patterns are used to select hypotheses in certain tactics such as iFrame and iLöb. Some of the selection patterns that the proof mode supports are:

- **H** : select the hypothesis named H.
- *%* : select the entire pure/Coq context. In the transitivity example presented above, we destruct some of our hypotheses into pure/Coq context so that we can use them later in our subproofs. We do this by placing % in front of the hypothesis name as seen in the tactic: iDestruct "Hxy" as %Hxy. This places the hypothesis into our Coq context as seen in Figure 15.

```
x, y, z :  loc
Φ :  val → iPropI Σ
Hxy :  x = y
```

```
(1/1)
"Hyz" : ⌜y = z⌝
"HΦ" : ▷ (⌜x = z⌝ -∗ Φ #())
------------------------------------∗
WP trans #x {{ v, Φ v }}
```

Figure 15. The result of using the tactic iDestruct "Hxy" as %Hxy to place hypothesis "Hxy" into the Coq context.

- # : select the entire intuitionistic context.
- $_*$: select the entire spatial context, note that $_*$ is the unicode symbol $_*$ and NOT the regular asterisk *.

6.2.4 Applying hypotheses and lemmas

Many of the IPM tactics can take both hypotheses and lemmas and allow one to instantiate universal quantifiers and implications or wands with these hypotheses and lemmas. We consider these proof mode terms (pm_trm). Some of the tactics in this category include:

- **iExact "H"** : this will finish the current goal of the proof if the conclusion matches the hypothesis H.
- **iApply pm_tm** : this will match the conclusion of the current goal against the conclusion of the pm_trm and generates goals for the premises of pm_trm. In the transitivity example presented above, we eventually have the goal of "ϕ#()" as seen in Figure 16. We see that this matches the right hand side of our hypothesis "HΦ". In order to get rid of this goal we can use the tactic iApply "HΦ" which will apply the right hand side so that we are left with proving the left hand side of the hypothesis as seen in Figure 17.

```
(1/1)
"HΦ" : ⌜x = z⌝ -∗ Φ #()
--------------------------------------∗
Φ #()
```

Figure 16. The proof state when we have goal "ϕ#()".

```
(1/1)
--------------------------------------∗
⌜x = z⌝
```

Figure 17. The result of using the tactic iApply "HΦ" which will apply the right hand side of "HΦ" so that we are left with only proving the left hand side of "HΦ".

6.2.5 Context management

Some of the tactics in this category include:

- **iIntros** $(x_1 ... x_n)$ **"ipat1 ... ipatn"** : introduces universal quantifiers using Coq introduction patterns $x_1 ... x_n$ and implications/wands using proof mode introduction patterns ipat1 ... ipatn. In the transitivity example presented above, we introduce the hypotheses "Hxy" and "HΦ". This tactic unfolds our triple notation and introduces the resulting hypotheses into our spatial context.

```
x, y, z :  loc
Φ :  val → iPropI Σ
```

```
(1/1)
"Hxy" : ⌜x = y⌝ ∗ ⌜y = z⌝ ∗ ⌜x = x⌝
"HΦ" : ▷ (⌜x = z⌝ -∗ Φ #())
--------------------------------------∗
WP trans #x {{ v, Φ v }}
```

Figure 18. The result of using the tactic iIntros (Φ) "Hxy HΦ" to introduce hypothesis "Hxy" and "HΦ into the spatial context of our proof state.

- **iRename "H1" into "H2"** : renames the hypothesis H1 into H2. In the transitivity example presented above, we rename our hypothesis "Hxy" to be "Heverything" because it is a more fitting name. As can be seen in the proof view below, the only thing that changed after our tactic call was the name of our hypothesis.

```
(1/1)
"Hxy" : ⌜x = y⌝ ∗ ⌜y = z⌝ ∗ ⌜x = x⌝
"HΦ" : ▷ (⌜x = z⌝ -∗ Φ #())
--------------------------------------∗
WP trans #x {{ v, Φ v }}
```

Figure 19. Before using the tactic iRename "Hxy" into "Heverything" we have hypotheses "Hxy" and "HΦ".

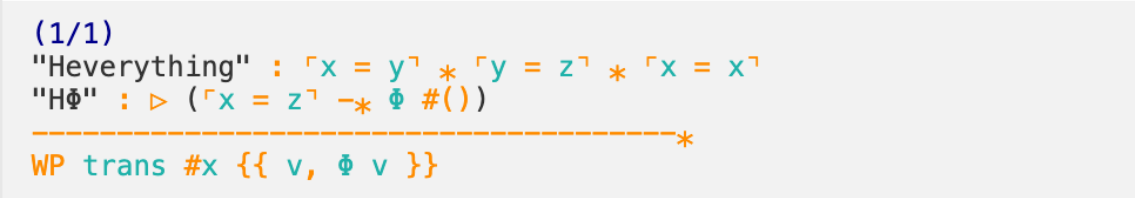

Figure 20. After using the tactic iRename "Hxy" into "Heverything" we have hypotheses "Heverything" and "HΦ".

- **iClear** $(x_1 ... x_n)$ **"selpat"** : clear the hypotheses given by the selection pattern selpat and the Coq level hypotheses variables $x_1 ... x_n$. In the transitivity example presented above, we notice that we have a hypothesis denoting "x=x". This is completely unneccessary for our proof, so we can use the tactic iClear in order to help keep our proof view easily readable. This tactic will merely get rid of the hypothesis/hypotheses you do not need. Note that this is most often used in large proofs when you have too many hypotheses and need to clear some to better visualize the goal at hand.

Figure 21. Before using the tactic iClear "Hxx" we have the hypothesis "Hxx".

Figure 22. After using the tactic iClear "Hxx" we no longer have the hypothesis "Hxx".

- **iPoseProof proof_mode_term as** $(x_1 ... x_n)$ **"ipat"** : puts the proof mode term into the context and destructs it using the introduction pattern ipat. This tactic is essentially the same as iDestruct, the difference being that the proof mode term is not thrown away if possible.

- **iAssert (P)%I with "specialization_pattern" as "H"** : generates a new subgoal P and adds the hypothesis P to the current goal as H. The specialization pattern specifies which hypotheses will be consumed by proving P. In the transitivity example presented above, we use this tactic multiple times. Let us look at the first case: iAssert (⌜x = z⌝)%I as %yz. When we use this tactic, it creates a new subgoal where we will need to prove what we assert. In this case we will prove the subgoal ⌜x = z⌝ as seen in Figure 23. After we prove this goal, it will be placed in our Coq context as the hypothesis "yz" denoted by %yz.

Figure 23. The result of using the tactic iAssert (⌜x = z⌝)%I as %yz. This tactic creates a new subgoal where we will need to prove ⌜x = z⌝.

6.2.6 Introduction of logical connectives

Some of the tactics in this category include:

- **iPureIntro** : turn a pure goal, typically of the form ⌜φ⌝, into a Coq goal. Here ⌜ ⌝ means the proposition φ will be placed into the pure Coq context, referred to in Figure 1 above the bold line. Notice in Figure 23 that our subgoal denoted (1/2) has the pure goal ⌜x = z⌝. We use this tactic to turn this into a Coq goal so we can then use the hypotheses we currently have in our Coq context for our proof. Figure 24 shows the state of our Coq context and subproof after the iPureIntro tactic.

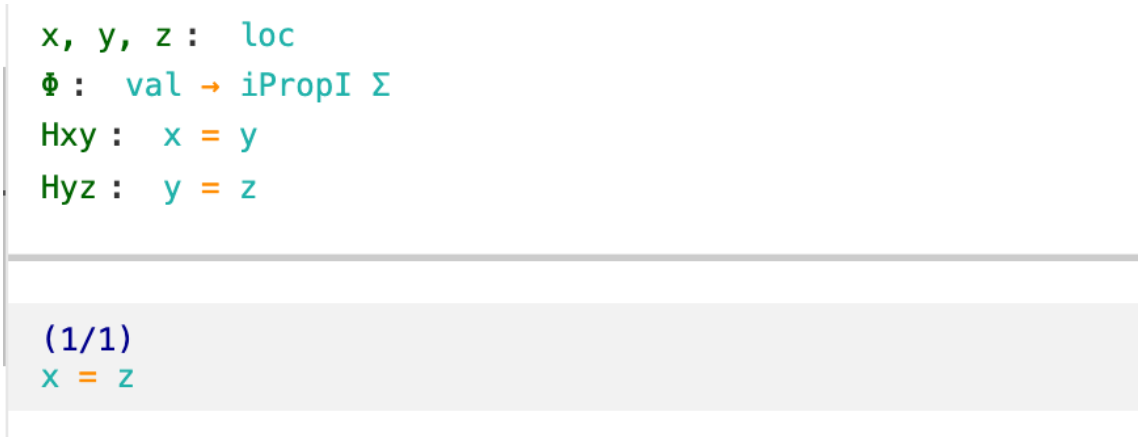

Figure 24. The result of using the tactic iPureIntro. ⌜x = z⌝ becomes a pure Coq goal so we can then use the hypotheses we currently have in our Coq context for our proof.

- **iLeft** : prove a disjunction $P \vee Q$ by proving the left side, P.
- **iRight** : prove a disjunction $P \vee Q$ by proving the right side, Q.
- **iSplitL "$H_1 ... H_n$"** : splits a conjunction $P * Q$ into two proofs. The hypotheses $H_1 ... H_n$ are used for the left conjunct and the remaining hypotheses are moved to the right conjunct. Intuitionistic hypotheses are always available in both proofs.
- **iSplitR "$H_0 ... H_n$"** : splits a conjunction $P * Q$ into two proofs. The hypotheses $H_1 ... H_n$ are used for the right conjunct. This is the symmetric tactic to iSplitL.
- **iExists "$t_1, ..., t_n$"** : provides a witness for an existential quantifier, $\exists x, ... t_1 ... t_n$.

6.2.7 Elimination of logical connectives

Some of the tactics in this category include:

- **iExFalso** : changes the goal to proving False.
- **iDestruct "H1" as ($x_1 ... x_n$) "H2"** : eliminates a series of existential quantifiers in hypothesis H1 using Coq introduction patterns $x_1 ... x_n$ and names the resulting hypothesis H2. There are many special cases of iDestruct, here we introduce only the most basic usage. In the transitivity example presented above, we have the hypothesis "Heverything" that contains three separating conjunctions as seen in Figure 25. The presence of these conjunctions tells us that we can further destruct this hypothesis, specifically into three individual hypotheses. We do this through the tactic iDestruct "Heverything" as "(Hxy & Hyz & Hxx)". This puts the three new hypotheses Hxy, Hyz, and Hxx into our proof state as seen in Figure 26.

```
(1/1)
"Heverything" : ⌜x = y⌝ ∗ ⌜y = z⌝ ∗ ⌜x = x⌝
"HΦ" : ▷ (⌜x = z⌝ −∗ Φ #())
--------------------------------------∗
WP trans #x {{ v, Φ v }}
```

Figure 25. We have the hypothesis "Heverything" that contains three separating conjunctions

```
(1/1)
"Hxy" : ⌜x = y⌝
"Hyz" : ⌜y = z⌝
"Hxx" : ⌜x = x⌝
"HΦ" : ▷ (⌜x = z⌝ −∗ Φ #())
--------------------------------------∗
WP trans #x {{ v, Φ v }}
```

Figure 26. The result of using the tactic the tactic iDestruct "Heverything" as "(Hxy & Hyz & Hxx)". This puts the three new hypotheses Hxy, Hyz, and Hxx into our proof state.

6.2.8 Separation logic-specific tactics

- **iFrame** ($t_1 ... t_n$) **"selpat"** : cancels the Coq terms or Coq hypotheses $t_1 ... t_n$ and Iris hypotheses given by selpat in the goal. The constructs of the selection pattern have the following meaning:
  - **%** : repeatedly frame hypotheses from the Coq context.
  - **#** : repeatedly frame hypotheses from the intuitionistic context.
  - ***** : frame as much of the spatial context as possible. Notice that framing spatial hypotheses makes them disappear, but framing Coq or intuitionistic hypotheses does not make them disappear. This tactic solves the goal if everything in the conclusion has been framed.

6.2.9 Modalities

Some of the tactics in this category include:

- **iModIntro** : introduces a modality in the goal. A deeper look into modalities occurred in Section 5.4. In the transitivity example presented above, we will come along the notation "|={ T }=>" which informs us that we have a modality as seen in Figure 27. We must introduce the modality into our goal, so we will use the tactic iModIntro as seen in Figure 28. This gets rid of our modality so we can continue with our proof.

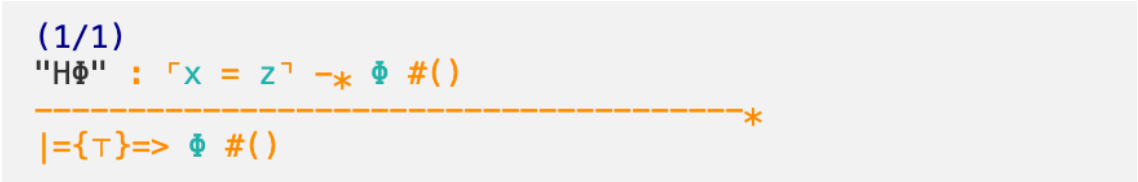

Figure 27. The notation "|={ T }=>" informs us that we have a modality.

Figure 28. The result of using the tactic iModIntro. This tactic introduced the modality into our goal.

- **iNext** : an alias of iModIntro (▷ ^_ _) which introduces the later modality. This eliminates a later in the goal and in exchange also strips one later from all the hypotheses.

6.2.10 Iris

- **iInv H as** (x_1...x_n) **"ipat"** : opens an invariant in hypothesis H. The result is destructed using the Coq introduction patterns x_1...x_n (for existential quantifiers) and then the proof mode introduction pattern ipat.

7 Iris Proof Set-up

When you begin a proof in Iris, declarations need to be made in a mindful order to ensure that you are not overriding other modules imported later. In order to make sure the most relevant declarations and notations always take priority, import dependencies from furthest to closest. In particular, import modules in the following order:

- Coq
- stdpp
- iris.algebra
- iris.bi
- iris.proofmode
- iris.base_logic
- iris.program_logic
- iris.heap_lang

We will see in Figure 29, for our example of a counter, that this order holds true for our dependencies. From these modules we export or import various files that will allow us to carry out our proof. For example, by requiring `From iris.proofmode Require Import tactics.`, we have access to files related to the interactive proof mode including the general tactics of the proof mode. For futher information, we refer readers to this work-in-progress document that explains further details about how Iris proofs are typically carried out in Coq.

```
From iris.proofmode Require Import tactics.
From iris.base_logic.lib Require Export invariants.

From iris.program_logic Require Export weakestpre.
From iris.heap_lang Require Import proofmode.

From iris.heap_lang Require Import notation lang.
From iris.heap_lang.lib Require Import par.
```

Figure 29. Example of dependencies that are required for our proof that a counter is correct.

Next, in order to complete our proof, we need to assume certain things about the instantiation of Iris. We section this portion of our file off by using the **Section** keyword.

Once we are in this section, we need to specify that the Iris instantiation has sufficient structure to manipulate heaps and has ghost states available. We will require the following lines for our example proof:

- `Context ‘{!heapG Σ}.` : This line states that we assume the Iris instantiation has sufficient structure to manipulate the heap which means it will allow us to use the points-to predicate (discussed in Section 4.1.1).
- `Context ‘{ !spawnG Σ}.` : This line states that we have the necessary ghost states to prove the parallel composition construct defined in terms of fork which will be discussed in Section 9.
- `Notation iProp := (iProp Σ).` : This line defines a shorthand notation for iProp Σ. The variable Σ has to do with the ghost state available and the type of Iris propositions (Prop) that depend on this Σ. Since we will refer to the same Σ throughout the whole file, we can define a shorthand notation that hides this Σ.
- `Context (N : namespace).` : This line provides the namespace parameter for our invariant (discussed in Section 5.3) throughout the whole file. We will need an invariant to share access to a location.

While we present these specifications for an Iris proof in terms of our specific example, everything will essentially be the same for most verifications. Therefore, it will help you create the start of your proof file no matter the specific problem at hand.

8 Case Study: Simple Counter

We will create one of the most simple program expressions to prove: *incr*. *incr* reads the value stored at heap location n into a variable $(l =!n)$ and then writes $n + 1$ back into the location. First we must define the program expression and program specs before we can prove they are correct. We determined these specifications in Section 5.7 and define them in Iris here. There are three different parts to defining and proving a program specification in Iris. We first have to define the program, denoted by the keyword **Definition**. When defining your program, it may be useful to use the Iris Coq library which defines many notations for programming language constructs, such as lambdas, allocation, accessing and so on. The complete list of notations can be found in the iris-coq repository.

In this example the # notation denotes an embeded literal, variable, or number as a value of the programming language HeapLang. Our definition is followed by our **Lemma** and **Proof**. The lemma we wish to prove here is that our program and the Hoare triple specifications is valid. We follow this lemma with the proof. In order to enter proof mode, we use the keyword **Proof** so that Coq knows we are ready to prove our lemma.

```
Definition incr : expr :=
λ: "l", let: "n" := !"l" in
        "l" <- "n" + #1.

Lemma incr_spec (ℓ : loc) (n : Z):
{{{ ℓ ↦ #n }}} incr #ℓ {{{RET #(); ℓ ↦ #(n+1)}}}.

Proof.
iIntros (Φ) "Hpt HΦ".
wp_pures. wp_load. wp_let. wp_op.
wp_store. iModIntro.
iApply "HΦ". iFrame.
Qed.
```

In order to start proving the specifications in Figure 30, we first must unfold the triple notation, $\{x \mapsto n\}\ incr\ \{x \mapsto n + 1\}$, using the tactic **iIntros (Φ) "Hpt HΦ"**. We see in Figure 31 that this tactic introduces two hypotheses into the global context, changing our goal in the process.

```
ℓ : loc
n : Z
```

```
(1/1)
{{{ ℓ ↦ #n }}} incr #ℓ {{{ RET #(); ℓ ↦ #(n + 1) }}}
```

Figure 30. The specifications of *incr* that we must prove.

```
ℓ : loc
n : Z
Φ : val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #n
"HΦ" : ▷ (ℓ ↦ #(n + 1) -∗ Φ #())
--------------------------------------∗
WP incr #ℓ {{ v, Φ v }}
```

Figure 31. The tactic iIntros (Φ) "Hpt HΦ" introduces two hypotheses into the spatial context; one from the precondition and one from the postcondition of the Hoare triple.

In our new goal, we now see the "WP" notation which stands for weakest precondition. We know that HeapLang provides us with a lot of tactics that facilitate stepping through a HeapLang program as part of proving a weakest precondition. Therefore, we can look to Section 6.1 to see which tactics we could use. We use the tactic **wp_pures** which will perform as many pure reduction steps as possible as seen in Figure 32.

```
ℓ : loc
n : Z
Φ : val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #n
"HΦ" : ℓ ↦ #(n + 1) -∗ Φ #()
--------------------------------------∗
WP let: "n" := ! #ℓ in #ℓ <- "n" + #1 {{ v, Φ v }}
```

Figure 32. The tactic wp_pures performs as many pure reduction steps as possible.

We now see not only a "WP" in our goal, but we also see a "let". Before we can deal with this "let" we must reduce the load operation. We use the tactic **wp_load** which automatically finds the points-to assertion in our spatial context. In this example, it finds the points-to assertion $\ell \mapsto \#n$. This means it will replace our ℓ with n which can be seen in Figure 33.

```
ℓ : loc
n : Z
Φ : val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #n
"HΦ" : ℓ ↦ #(n + 1) -∗ Φ #()
--------------------------------------∗
WP let: "n" := #n in #ℓ <- "n" + #1 {{ v, Φ v }}
```

Figure 33. The tactic wp_load finds the points-to assertion $\ell \mapsto \#n$ and replaces ℓ with n.

We can now do some reductions. First, we reduce our let-binding through the tactic **wp_let** as seen in Figure 34.

```
ℓ : loc
n : Z
Φ : val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #n
"HΦ" : ℓ ↦ #(n + 1) -∗ Φ #()
--------------------------------------∗
WP #ℓ <- #n + #1 {{ v, Φ v }}
```

Figure 34. The tactic wp_let reduces the let-binding.

Then, we want to reduce $\#n + \#1$ to $\#(n + 1)$ which can be done through the tactic **wp_op** which reduces unary or binary arithmetic operators as can be seen in Figure 35.

```
ℓ : loc
n : Z
Φ : val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #n
"HΦ" : ℓ ↦ #(n + 1) -∗ Φ #()
--------------------------------------∗
WP #ℓ <- #(n + 1) {{ v, Φ v }}
```

Figure 35. The tactic wp_op reduces $\#n + \#1$ to $\#(n + 1)$.

We will now want to reduce our store operation. We use the tactic **wp_store** which automatically finds the points-to assertion in our spatial context. Once again the points-to assertion it finds is $\ell \mapsto \#n$ which can now be replaced with $\ell \mapsto \#(n + 1)$ as seen in Figure 36.

```
ℓ : loc
n : Z
Φ : val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #(n + 1)
"HΦ" : ℓ ↦ #(n + 1) -∗ Φ #()
--------------------------------------∗
|={⊤}=> Φ #()
```

Figure 36. The tactic wp_store reduces the store operation. Find the points-to assertion $\ell \mapsto \#n$ and replaces it with $\ell \mapsto \#(n + 1)$.

After we apply this tactic, we see the notation "|={ T }=>" which informs us that we have a modality. We can refer to Section 6.2.9 to determine which tactic we need to use next. Here we must introduce the modality into our goal, so we will use the tactic **iModIntro** as seen in Figure 37.

```
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #(n + 1)
"HΦ" : ℓ ↦ #(n + 1) -∗ Φ #()
------------------------------------∗
Φ #()
```

Figure 37. The tactic iModIntro introduces the modality into our goal.

We are left with "$\Phi\ \#()$" in our goal. If we look at our hypothesis "$H\Phi$" we see that we have both sides of the separating conjunction. The left side matches the hypothesis "Hpt" and the right hand side is our current goal. We can apply hypothesis "$H\Phi$" to our goal in order to reduce "$H\Phi$" to just the left side, $\ell \mapsto \#(n+1)$. We do this through the tactic **iApply "$H\Phi$"**. This tactic matches the conclusion of the current goal against the conclusion of $H\Phi$ and generates goals for the premises of $H\Phi$.

```
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
```

```
(1/1)
"Hpt" : ℓ ↦ #(n + 1)
------------------------------------∗
ℓ ↦ #(n + 1)
```

Figure 38. The tactic iApply "$H\Phi$" matches the conclusion of the current goal against the conclusion of $H\Phi$ and generates goals for the premises of $H\Phi$. Our new proof is $\ell \mapsto \#(n+1)$.

We see in Figure 38 that this means our new proof will be $\ell \mapsto \#(n+1)$. This new goal is the same as hypothesis "Hpt" which means we can easily finish this proof by cancelling out this hypothesis. The tactic **iFrame** allows us to cancel Coq hypothesis "Hpt" and finish our proof. We see the phrase **"No more subgoals."** which tells us we have completed the proof and can end it with the tactic **Qed**. As discussed in Section 3.1.1.1, **Qed** is how we inform Coq that our proof is done and exit proof mode.

9 Case Study: Parallel Counter

We will explain parts of this parallel increment example, however, to find the full code and explanation, refer to the Iris Project. Before we dive into the example, let us provide some brief background knowledge on parallel threads. In the program we are going to define, we will use the commonly used parallel composition syntax $(e_1||e_2)$ which denotes forking two threads that run e_1 and e_2 respectively and waiting until they complete. Parallel composition can be defined in terms of **fork** and CAS. [4] We will not go into the details of either term in this guide as it is not our main focus. We now define our terms and the invariant we are going to use as the following:

```
Definition counter (ℓ : loc) : expr := #ℓ <- !#ℓ + #1.

Definition counter_inv (ℓ : loc) (n : Z) :
iProp := (∃ (m : Z), ⌜(n ≤ m)%Z⌝ ∗ ℓ ↦ #m)%I.
```

The body of our invariant is defined as *counter_inv*. Note that we use **"%I"** to tell Coq to parse the logical formula as an Iris assertion; this means it will interpret the connectives as Iris connectives. We also use ⌜...⌝ which is used to embed Coq propositions as Iris assertions. In this example, we make "≤" an Iris assertion. In simple terms, this embedding means that the embedded assertion either holds for all resources or for none. We now define the specifications of the parallel counter we wish to prove.

```
Lemma parallel_counter_spec (ℓ : loc) (n : Z):
    {{{ ℓ ↦ #n }}} (counter ℓ) ||| (counter ℓ) ;; !#ℓ
            {{{m, RET #m; ⌜(n ≤ m)%Z⌝ }}}.
```

Here we will describe the tactics that effect our invariant. We first want to allocate an invariant (in namespace N) and transfer the points-to-predicate into it. This can be achieved by using the tactic **inv_alloc**. The allocation of an invariant involves the fancy update modality which means we also must use the **iMod** tactic around the lemma. The **iMod** tactic knows about the structural rules of the update modalities and the interaction of the update modality and the weakest precondition assertion and therefore it will automatically remove the modality as much as possible. Our full tactic for this purpose becomes **iMod (inv_alloc N _ (incr_inv ℓ n) with "[Hpt]") as "#HInv"**. The tactic **inv_alloc** takes three parameters, the namespace in which to allocate, the mask used on the fancy update modality (not important for our purposes so here we leave it implicit by using _), and the body of the invariant to be allocated. We see this tactic also contains the "with" construct. This tells **iMod** which of the assumptions are to be

[4] cite book

used to satisfy the assumptions of the **inv_alloc** rule. Finally, the last part of this tactic is "as "#Hinv"". This tells the **iMod** tactic to name the conclusion of the **inv_alloc** rule "Hinv" and add it to the persistent hypotheses; the # notation is used to denote persistent hypotheses.

In general terms of the structure of this proof, we would have three subgoals. We would have two proofs to show each thread does the correct thing, and a third goal to show that the combined conclusion of the two threads implies the desired conclusion. Let us first consider the proof that the first thread is correct. The expression is not atomic, so we cannot open the invariant immediately. Thus we must first use the bind rule to reshape it, then we can open the invariant. We can see the state of our proof in Figure 39 right before we open our invariant. The next portion of this proof will show how to open and close this invariant.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
-----------------------------------□
WP ! #ℓ {{ v, WP #ℓ <- v + #1 {{ _, True }} }}
```

Figure 39. The proof state before we open the invariant. The last tactic used before this process was wp_bind (!#ℓ)%E.

We can open the invariant to read a value stored at location ℓ. We open an invariant by using tactic **iInv**. This tactic takes a namespace as the first argument and two named assertions. There are two parts of the **inv_open** rule because there are two parts to the conclusion. Therefore, we use the full tactic **iInv N as "H" "Hclose"** as can be seen in Figure 40.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
"H" : ▷ counter_inv ℓ n
"Hclose" : ▷ counter_inv ℓ n ={⊤ ∖ ↑N,⊤}=∗ emp
──────────────────────────────────────∗
WP ! #ℓ @ ⊤ ∖ ↑N {{ v, |={⊤ ∖ ↑N,⊤}=> WP #ℓ <- v + #1 {{ _, True }} }}
```

Figure 40. We use tactic iInv N as "H" "Hclose" to open the invariant so that we have access to the resource and can read the value of it.

We now can read the value since we have the resources. However, if we look at the *incr_inv* assertion, we see we need to eliminate our existential "H" to get a points-to predicate before we can read the memory location. We can do this through the **iDestruct** tactic which will eliminate the existential quantifier in the hypothesis. We will not go into details here as to the rest of the syntax for this tactic, but this tactic will provide us with a points-to assertion in context as seen in Figure 41.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
m :  Z
H :  (n ≤ m)%Z
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
"Hpt" : ℓ ↦ #m
"Hclose" : ▷ counter_inv ℓ n ={⊤ ∖ ↑N,⊤}=∗ emp
──────────────────────────────────────∗
WP ! #ℓ @ ⊤ ∖ ↑N {{ v, |={⊤ ∖ ↑N,⊤}=> WP #ℓ <- v + #1 {{ _, True }} }}
```

Figure 41. We use tactic iDestruct "H" as (m) ">[% Hpt]" to eliminate the existential quantifier in the hypothesis.

We can then use the **wp_load** tactic to read the memory location in Figure 42.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
m :  Z
H :  (n ≤ m)%Z
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
"Hpt" : ℓ ↦ #m
"Hclose" : ▷ counter_inv ℓ n ={⊤ ∖ ↑N,⊤}=∗ emp
──────────────────────────────────────∗
|={⊤ ∖ ↑N,⊤}=> WP #ℓ <- #m + #1 {{ _, True }}
```

Figure 42. We use tactic wp_load to read the memory location.

Now that we have read our value, we must close the invariant. We use the "Hclose" assertion that we got when we opened the invariant to now close it. We close the invariant by transferring the points-to predicate back. Closing the invariant involves manipulation of fancy update modalities so we must use the **iMod** tactic. We do not care about the conclusion of the "Hclose" assertion in this case, so we will use the "_" notation to state we can ignore it. Our full tactic will be **iMod("Hclose" with "[Hpt]") as "_"** whose results can be seen in Figure 43.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
m :  Z
H :  (n ≤ m)%Z
```

```
(1/2)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
"Hpt" : ℓ ↦ #m
──────────────────────────────────────∗
▷ counter_inv ℓ n
```

```
(2/2)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
|={⊤}=> WP #ℓ <- #m + #1 {{ _, True }}
```

Figure 43. We use tactic iMod("Hclose" with "[Hpt]") as "_" to close the invariant.

As can be seen in Figure 43 above, we now have a subgoal we must prove before we are done with closing our invariant. Our subgoal, denoted by (1/2), starts with the later modality ▷ which tells us we need to use the tactic **iNext**. This tactic will introduce the later modality, eliminating it from our goal as seen in Figure 44.

```
N : namespace
ℓ : loc
n : Z
Φ : val → iPropI Σ
m : Z
H : (n ≤ m)%Z
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────□
"Hpt" : ℓ ↦ #m
──────────────────────────────────∗
counter_inv ℓ n
```

Figure 44. We use tactic: iNext to introduce the later modality and eliminate it from our goal.

We must also introduce m into our goal which is possible through the tactic **iExists** m since there is an existential quantifier in our invariant. The tactic **iExists** provides a witness for an existential quantifier, in this case (∃ (m : Z)) as seen in Figure 45.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
m :  Z
H :  (n ≤ m)%Z
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
"Hpt" : ℓ ↦ #m
──────────────────────────────────────*
⌜(n ≤ m)%Z⌝ ∗ ℓ ↦ #m
```

Figure 45. We use tactic: iExists m to provide a witness for the existential quantifier (∃ (m : Z)).

At this point we see we have half of the separating conjunction, meaning we can use the tactic **iFrame** to cancel the right side of the Coq term in the goal as done in Figure 46.

```
N :  namespace
ℓ :  loc
n :  Z
Φ :  val → iPropI Σ
m :  Z
H :  (n ≤ m)%Z
```

```
(1/1)
"HInv" : inv N (counter_inv ℓ n)
──────────────────────────────────────□
⌜(n ≤ m)%Z⌝
```

Figure 46. We use tactic: iFrame to cancel the right side of the Coq term in the goal.

We can see we are almost finished with this subproof, however our goal is wrapped in ⌜...⌝ which means it is in pure Coq context and we must introduce a pure goal. We do this through the notation "!%" and the tactic **iIntros**, giving us the full tactic **iIntros "!%"**. We then see our goal in Figure 47 perfectly matches the Coq hypothesis "H" in our global context.

```
N : namespace
ℓ : loc
n : Z
Φ : val → iPropI Σ
m : Z
H : (n ≤ m)%Z
```

```
(1/1)
(n ≤ m)%Z
```

Figure 47. We use tactic iIntros "!%" to introduce $\ulcorner(n \leq m)\%Z\urcorner$ as a pure Coq goal. Note that the tactic iPureIntro would also work here.

We are able to use the tactic **auto** which will automatically find this connection and finish our subproof for us. We know this is the case when we see the phrase **"There are unfocused goals."**. This means we can continue our main proof. All of these tactics were described one after each other because they could be linked into one big tactic. We can use ";" to string together tactics as follows:**iNext; iExists m; iFrame; iIntros "!%"; auto.** This one tactic will have the same effect as all 4 tactics being individually called.
This fully concludes the opening and closing of the invariant which may have to be done multiple times throughout our proof. In total, this process required the chunk of code below. While this was specific to our parallel increment example, the tactics used to open invariants in most cases will look similar.

```
iInv N as "H" "Hclose".
iDestruct "H" as (m) ">[% Hpt]".
wp_load.
iMod ("Hclose" with "[Hpt]") as "_".
  { iNext. iExists m. iFrame. iIntros "!%". auto. }
```

10 Case Study: Bank

We will explain parts of a bank example, however, we want to note that this is a simplified version of the bank implemented in: A brief introduction to Iris. Our version does not include locks in our bank for simplicity. But, we provide a deeper explanation into the Iris tactics being used, especially in terms of ghost states. For interested readers, we highly suggest working through the full, more advanced code in that guide, upon total understanding of this version of the bank.

Before we dive into the definitions for this example, we need to add different declarations than we presented in Section 7. Here we must take into account our invariants and ghost states, so we add the following dependencies to our proof:

```
From iris.algebra Require Import lib.excl_auth.
From iris.heap_lang Require Import proofmode notation.
From iris.base_logic.lib Require Export invariants.
Set Default Proof Using "All".
```

We also need to specify the cameras/RAs used as ghost states in this proof. To do so we add the following line to go with our other contexts:

```
Context `{ inG Σ (authR (optionUR (exclR ZO)))} `{OfeDiscrete ZO}.
```

Now we can begin defining our program. We will need two main parts for this program: the bank with our accounts and the ability to transfer money from one account to another. Once we have these, we will also need to formulate our invariants. We will need one invariant to be associated with the account (account_inv), it will tie the account balance to a ghost variable and take the location where the account balance is stored. The other invariant will be associated with the bank (bank_inv), it will hold the fragments of the account balances and states that the account balances sum to zero.

First, let us define our bank. Our function *bank* will construct a bank with two accounts, both with a zero balance:

```
Definition bank: val :=
 λ: <>,
 let: "a_bal" := ref #0 in
 let: "b_bal" := ref #0 in
 ( "a_bal" , "b_bal").
```

We now define our *transfer* function. The function *transfer* will move money from the first to the second account. Note that we do not check that there is enough money in the account before transferring the money, meaning negative balances are allowed. Here we verify only that this function is safe.

```
Definition transfer: val :=
 λ: "bank" "amt",
 let: "a" := Fst "bank" in
 let: "b" := Snd "bank" in
 Snd "a" <- !(Snd "a") - "amt";;
 Snd "b" <- !(Snd "b") + "amt";;
 #().
```

Now that we have the basic setup of our bank, we can define our invariants as described above:

```
Definition bank_inv (γ1 γ2: gname): iProp :=
  (∃ (bal1 bal2: Z),
   own γ1 (●E bal1) *
   own γ2 (●E bal2) *
   ⌜(bal1 + bal2)%Z = 0⌝)%I.

Definition account_inv γ bal_ref : iProp :=
  ∃ (bal: Z), bal_ref ↦ #bal * own γ (◯E bal).
```

In our invariant we introduce ghost variables. In order to reason about these later, we must also introduce the function *ghost_var_alloc*. In these ghost states, *a* represents an element of arbitrary type. The ghost state **own γ_1 (●E bal)** represents "authoritative ownership" and the ghost state **own γ (◯E bal)** represents "fragmentary ownership", however, both have exclusive ownership meaning they are symmetric. In general terms, the state that has authoritative ownership will go to the global invariant and the state that has fragmentary ownership will be handed out to other invariants (such as lock invariants). By using the function *ghost_var_alloc* we can allocate the pair of ghost states. We will not explain the function *ghost_var_alloc* any further, but we will show how to use it.

```
Lemma ghost_var_alloc `{inG Σ (authR (optionUR (exclR A)))}
    `{OfeDiscrete A} (a: A) :
          ⊢ |==> ∃ γ, own γ (●E a) * own γ (◯E a).
Proof.
  iMod (own_alloc (●E a · ◯E a)) as (γ) "[H1 H2]".
  { apply excl_auth_valid. }
  iModIntro. iExists γ. iFrame.
Qed.
```

The last piece we need to define before we can prove our theorem is a way to tie together each account and the bank. We do this through *is_bank*:

```
Definition is_bank (b: val): iProp :=
  (∃ (acct1 acct2: val) (γ1 γ2: gname),
  ⌜ b = (acct1, acct2)%V⌝ *
  is_account acct1 *
  is_account acct2 *
  inv N (bank_inv γ1 γ2)).
```

The proof we will go through in detail for this example is the function, *bank*. The function *bank* has to create all of the ghost states, invariants, and argue all of these initially hold. Therefore, we will explain this code in great detail to give some insight on how to use ghost states and the tactics related to them.

We need to define our program specifications for the theorem we are going to prove, *wp_bank*. Similar to our other examples, we can do this through a Hoare triple. Our Hoare triple for this program will be $\{True\}\ bank\ \{\exists b, RET\ b; is\_bank\ b\}$. Note that here, *RET b* means that we return b in the postcondition. We can write this in Iris logic as follows:

```
Theorem bank_spec :
  {{{ True }}}
    bank #()
  {{{ b, RET b; is_bank b }}}.
```

We start off our proof with two simple tactics. We first unfold our Hoare Triple through the tactic **iIntros (Φ) "_ HΦ"** and then we want to unfold the function *bank* through the tactic **wp_rec**. This leaves us with the proof state seen below in Figure 48.

```
Φ :  val → iPropI Σ
```

```
(1/1)
"HΦ" : ∀ b : val, is_bank b -∗ Φ b
--------------------------------------∗
WP let: "a_bal" := ref #0 in let: "b_bal" := ref #0 in ("a_bal", "b_bal") {{ v, Φ v }}
```

Figure 48. Our proof state after unfolding our Hoare triple and the function *bank*.

In our goal we see we have two references, "a_bal" := ref #0 and "b_bal" := ref #0, one for each account. We want to pull these out of our goal and into our spatial context so we can use them. In order to do this, we can use the HeapLang tactic **wp_alloc l as "H"**. This tactic reduces the allocation (reference) instruction and allows us to put it at a new location. Our full tactic here will be **wp_alloc a_ref as "Ha"**. This tells Iris we are taking the reference of "a_bal" (since it occurs first in the goal) and placing it at location "a_ref" in our Coq context and naming its points-to assertion "Ha" in our spatial context. Similarily, we will use tactic **wp_alloc b_ref as "Hb"** to do the same for our second reference "b_bal". This brings us to the proof state seen in Figure 49.

```
Φ :  val → iPropI Σ
a_ref, b_ref :  loc
```

```
(1/1)
"HΦ" : ∀ b : val, is_bank b -∗ Φ b
"Ha" : a_ref ↦ #0
"Hb" : b_ref ↦ #0
--------------------------------------∗
WP let: "b_bal" := #b_ref in (#a_ref, "b_bal") {{ v, Φ v }}
```

Figure 49. Our proof state after reducing our references for each account balance through the tactics wp_alloc a_ref as "Ha" and wp_alloc b_ref as "Hb".

We are now at the point of the proof where we need to introduce our ghost variables. We will execute the ghost variable change using the function *ghost_var_alloc* and destruct it as ghost variable γ_1 with names "Hown1" and "Hγ_1". Here "Hown1" and "Hγ_1" name the two resources **own γ (●E a)** and **own γ (○E a)** respectively. As mentioned earlier, a represents an element of arbitrary type, in our specific example a will be replaced with 0. In our function, we can allocate a new ghost variable, under an update modality because it requires modifying the global ghost state as seen below (Chajed, 2020).

As we already discussed above, we need to execute the ghost variable change. To do so, we use the tactic **iMod (ghost_var_alloc (0: ZO)) as (γ_1) "(Hown1 & Hγ_1)"** which introduces **own γ_1 (●E 0)** and **own γ_1 (○E 0)** as seen in Figure 50. Similarly, we do the exact same thing for parallel results for our second ghost variable γ_2.

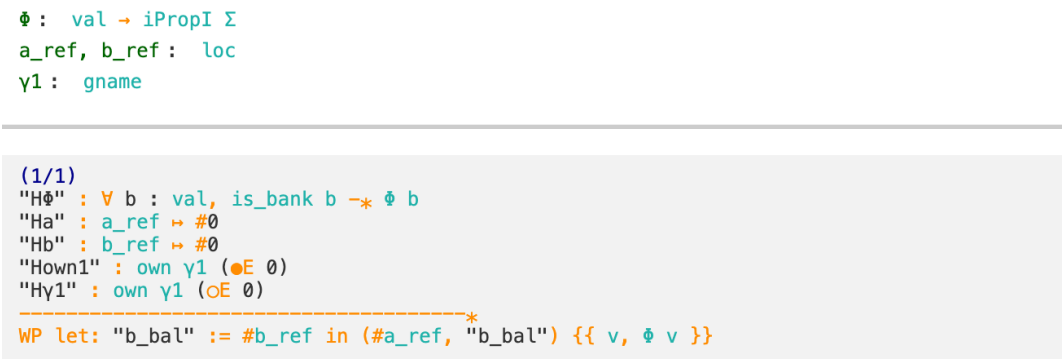

Figure 50. Proof state after using tactic iMod (ghost_var_alloc (0: ZO)) as (γ_1) "(Hown1 & Hγ_1)". Similarly, we do the exact same thing for parallel results for our second ghost variable γ_2.

Now that we have introduced our ghost states, we can allocate the *bank_inv* invariant. Recall that in *bank_inv* we say the balances of the two accounts sums to zero. We will show that this statement initially holds true. In order to allocate our invariant we must use the tactic **inv_alloc**. Remember from the parallel increment example that **inv_alloc** takes three parameters, the namespace in which to allocate, the mask used on the fancy update modality (not important for our purposes so here we leave it implicit by using _), and the body of the invariant to be allocated. As stated before, the allocation of an invariant also involves the fancy update modality which means we must use the **iMod** tactic around the lemma. Therefore, our full tactic results **iMod (inv_alloc N _ (bank_inv $\gamma_1\gamma_2$) with "[Hown1 Hown2]") as "Hinv"**. The resulting proof state seen in Figure 51 requires us to finish off the subgoal this creates. This subproof will be only five tactics, all of which we have explained in detail elsewhere in this guide. We suggest attempting this small proof on your own, before referring to the code below.

```
Φ : val → iPropI Σ
a_ref, b_ref : loc
γ1, γ2 : gname
```

```
(1/2)
"Hown1" : own γ1 (●E 0)
"Hown2" : own γ2 (●E 0)
--------------------------------------∗
▷ bank_inv γ1 γ2
```

```
(2/2)
"HΦ" : ∀ b : val, is_bank b -∗ Φ b
"Ha" : a_ref ↦ #0
"Hb" : b_ref ↦ #0
"Hγ1" : own γ1 (◯E 0)
"Hγ2" : own γ2 (◯E 0)
"Hinv" : inv N (bank_inv γ1 γ2)
--------------------------------------∗
WP let: "b_bal" := #b_ref in (#a_ref, "b_bal") {{ v, Φ v }}
```

Figure 51. Proof state after using tactic iMod (inv_alloc N _ (bank_inv $\gamma_1\gamma_2$) with "[Hown1 Hown2]") as "Hinv".

The rest of the proof is straightforward, using many of the tactics we have already explained in great detail. The full code for the proof is posted below, however we will not go into detail about each tactic. While working out the rest of this example, we highly suggest referring to Section 6 to see all of the tactics available to you.

```
Proof.
iIntros (Φ) "_ HΦ".
wp_rec.
wp_alloc a_ref as "Ha".
wp_alloc b_ref as "Hb".
iMod (ghost_var_alloc (0: ZO)) as (γ1) "(Hown1 & Hγ1)".
iMod (ghost_var_alloc (0: ZO)) as (γ2) "(Hown2 & Hγ2)".
iMod (inv_alloc N _ (bank_inv γ1 γ2)
with "[Hown1 Hown2]") as "Hinv".
{ iNext. iExists _, _. iFrame. iPureIntro. auto. }
wp_pures.
iApply "HΦ".
iExists _. iExists _. iFrame. iModIntro. iExists γ1. iExists γ2.
iSplit. first eauto. iSplitL "Ha Hγ1".
- iExists _. unfold account_inv. iSplitR. eauto with iFrame.
  iExists γ1. iExists _. iFrame.
- iSplitL "Hb Hγ2". iExists _. unfold account_inv. iSplitR. eauto
  with iFrame. iExists γ2. iExists _. iFrame. done.
Qed.
```

11 Related Work

After working through this guide, we assume readers will want further Iris and Coq resources, a few of which we will mention here.

- To further your knowledge on Iris tactics, we refer readers to the Iris documentation.
- To further your knowledge on setting up Iris proofs in Coq, we refer readers to the Iris Proof Guide.
- To further your knowledge on HeapLang tactics, we refer readers to the HeapLang documentation.
- For more help with installing Iris on your device, refer to this in-depth guide.
- For help inputting and outputting unicode characters throughout Iris, refer to this guide.
- For more help with installing opam on your device (before you install Iris), refer to the opam documentation.
- To further your knowledge on Coq and Coq tactics, we refer readers to the Coq documentation.
- For more help with installing Coq on your device, refer to the Coq Proof Assistant documentation.
- For more details on separation logic in relation to Iris and concurrent data structures, we refer readers to the following papers: Iris from the ground up, Lecture Notes on Iris: Higher-Order Concurrent Separation Logic, and The Essence of Higher-Order Concurrent Separation Logic.
- For more proof examples, many of which are advanced, refer to the Iris project note files.
- For other Iris and Coq guides that build off of this one, we suggest the following: A brief introduction to Iris, the Iris Project Lecture Notes, and once again the Lecture Notes on Iris: Higher-Order Concurrent Separation Logic

References

Lars Birkedal and Ales Bizjak. 2020. Lecture notes on iris: Higher-order concurrent separation logic. Aarhus University.

James Brotherston, Diana Costa, Aquinas Hobor, and John Wickerson. 2020. Reasoning over permissions regions in concurrent separation logic. *Computer Aided Verification*, 12225:203–224.

Tej Chajed. 2020. A brief introduction to iris.

Stéphane Demri and Morgan Deters. 2015. Separation logics and modalities: A survey. *Journal of Applied Non-Classical Logics*.

Michael J. C. Gordon. 2014. Hoare logic.

Rajeev Goré. 2016. Weakest precondition calculus.

C.A.R. Hoare. 1969. An axiomatic basis for computer programming. *Communications of the ACM*, 12.

Inria, CNRS, and Contributors. 2020. Coq reference manual.

Ralf Jung. 2021a. coq_intro_example_1.v.

Ralf Jung. 2021b. Heaplang overview.

Ralf Jung. 2021c. Tactic overview.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up. *Functional Programming*.

Robbert Krebbers. 2017. Demonstration of the iris separation logic in coq. Delft University of Technology, The Netherlands.

Robbert Krebbers, Ralf Jung, Aleš Bizjakand Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017. The essence of higher-order concurrent separation logic. *European Symposium on Programming*.

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. (In Press). *Automated Verification of Concurrent Search Structures*. Morgan and Claypool Publishers.

Peter W. O'Hearn. 2012. A primer on separation logic (and automatic program verification and analysis). *Software Safety and Security; Tools for Analysis and Verification*, 33:286–318.

Jung Ralf, Swasey David, Sieczkowski Filip, Svendsen Kasper, Turon Aaron, Birkedal Lars, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM*, pages 637–650.

Jan Schwinghammer, Lars Birkedal, Bernhard Reus, and Hongseok Yang. 2009. Nested hoare triples and frame rules for higher-order store.

Spies Simon, Gäher Lennard, Gratzer Daniel, Tassarotti Joseph, Krebbers Robbert, Dreyer Derek, and Birkedal Lars. 2020. The transfinite iris documentation.

Appendix

We include a brief reference for inputting unicode symbols in Vscde with the 'Generic Input Method' extension installed and configured. More details on how to complete these steps can be found in the general unicode guide for Iris.

Unicode Symbols for Iris			
Symbol	Name	Code Point	Vscode Generic Input Method (prefix with \)
¬	NOT SIGN	0xac	not
↑	UPWARDS ARROW	0x2191	uparrow
→	RIGHTWARDS ARROW	0x2192	to
↔	LEFT RIGHT ARROW	0x2194	iff
↦	RIGHTWARDS ARROW FROM BAR	0x21a6	mapsto
∀	FOR ALL	0x2200	all
∃	THERE EXISTS	0x2203	ex
∅	EMPTY SET	0x2205	empty
∈	ELEMENT OF	0x2208	in
∉	NOT AN ELEMENT OF	0x2209	notin
∖	SET MINUS	0x2216	-
∗	ASTERISK OPERATOR	0x2217	star
∘	RING OPERATOR	0x2218	comp
∧	LOGICAL AND	0x2227	and
∨	LOGICAL OR	0x2228	or
∩	INTERSECTION	0x2229	cap
∪	UNION	0x222a	cup
≠	NOT EQUAL TO	0x2260	neq
≡	IDENTICAL TO	0x2261	==
≤	LESS THAN OR EQUAL TO	0x2264	leq
≫	MUCH GREATER THAN	0x226b	
≼	PRECEDES OR EQUAL TO	0x227c	incl
⊆	SUBSET OF OR EQUAL TO	0x2286	subseteq

Unicode Symbols for Iris			
Symbol	Name	Code Point	Vscode Generic Input Method (prefix with \)
$\vdash$	RIGHT TACK	0x22a2	ent
$\top$	DOWN TACK	0x22a4	top
$\cdot$	DOT OPERATOR	0x22c5	mult
$\ulcorner$	TOP LEFT CORNER	0x231c	lc
$\urcorner$	TOP RIGHT CORNER	0x231d	rc
$\square$	WHITE SQUARE	0x25a1	box
$\triangleright$	WHITE RIGHT POINTING TRIANGLE	0x25b7	later
$\diamond$	WHITE DIAMOND	0x25c7	diamond
$\bullet$	BLACK CIRCLE	0x25cf	auth
$\bigcirc$	LARGE CIRCLE	0x25ef	frag
$\checkmark$	CHECK MARK	0x2713	valid
$\models\!\!\Longrightarrow$	RIGHTWARDS DOUBLE ARROW	0x2907	
½	VULGAR FRACTION ONE HALF	0xbd	
$_m$	LATIN SUBSCRIPT SMALL LETTER m	0x2098	_m
ö	LATIN LETTER O WITH DIAERESIS	0xf6	"o
γ	GREEK SMALL LETTER GAMMA	0x3b3	gamma
ϵ	GREEK SMALL LETTER EPSILON	0x3b5	epsilon
λ	GREEK SMALL LETTER LAMBDA	0x3bb	lambda
Σ	GREEK CAPITAL LETTER SIGMA	0x3a3	Sigma
φ	GREEK SMALL LETTER PHI	0x3c6	phi
Φ	GREEK PHI SYMBOL	0x3d5	
ϕ	GREEK CAPITAL LETTER PHI	0x3a6	Phi
ψ	GREEK CAPITAL LETTER PSI	0x3a8	Psi