Lecture Notes on
Iris: Higher-Order Concurrent Separation Logic

Lars Birkedal and Aleš Bizjak

Aarhus University {birkedal,abizjak}@cs.au.dk

December 14, 2017

With contributions by: Kristoffer Just Andersen (Aarhus University), Johan Bay (Aarhus University), Daniel Gratzer (Carnegie Mellon University), Mathias Høier (Aarhus University), Robbert Krebbers (Delft University of Technology), Marit Edna Ohlenbusch (Aarhus University).

Contents

1 Introduction **3**

2 Programming Language **3**

3 The logic of resources **4**
3.1 Propositions and entailment . . . 6
3.2 Rules for separating conjunction and magic wand . . . 9
3.3 Basic mathematical constructions in Iris . . . 11

4 Separation logic for sequential programs **11**
4.1 Derived rules for Hoare triples, and examples . . . 20
4.2 Abstract data types . . . 28
4.3 Abstract data types and ownership transfer: a stack module . . . 29
4.4 Case Study: foldr . . . 32

5 The later modality **38**
5.1 Stronger rules for Hoare triples . . . 40
5.2 Recursively defined predicates . . . 41

6 The always modality **42**

7 Invariants and ghost state **45**
7.1 The par construct . . . 45
7.2 Invariants . . . 46
7.3 A peek at ghost state . . . 52
7.4 Ghost state . . . 55
7.5 Compare and set primitive . . . 65
7.6 Examples . . . 65
7.7 Authoritative resource algebra: counter modules . . . 73
7.8 Case Study: Invariants for Sequential Programs . . . 78

8 First steps towards the base logic **79**
8.1 Weakest precondition . . . 79
8.2 Fancy update modality . . . 82
8.2.1 The fancy update modality and weakest precondition . . . 84
8.3 Timeless propositions . . . 88
8.4 Invariant namespaces . . . 90

9 Iris Proof Mode in Coq **92**

10 Case Study: Stacks with Helping **94**
10.1 Mailboxes for Offers . . . 94
10.2 The Implementation of the Stack . . . 96
10.3 A Bag Specification . . . 97
10.4 Verifying Offers . . . 98
10.5 Verifying Mailboxes . . . 98
10.6 Verifying Stacks . . . 99

11 The Essence of Iris **100**

12 Semantics **100**

13 Logical Atomicity **100**

14 Types and Abstraction: Logical Relations in Iris **100**

Preface

These lecture notes are intended to serve as an introduction to Iris, a higher-order concurrent separation logic framework implemented and verified in the Coq proof assistant.

Iris has been developed over several years in joint research involving the Logic and Semantics group at Aarhus University, led by Lars Birkedal, and the Foundations of Programming Group at Max Planck Institute for Software Systems, led by Derek Dreyer. Lately, the development has involved several other international research groups, in particular the group of Robbert Krebbers at TU Delft.

The main research papers describing the Iris program logic framework are three conference papers [4, 2, 5] and a longer journal paper with more details on the semantics and the latest developments of the logic [3]. These papers, and several other Iris related research papers, can all be found on the Iris Project web site:

iris-project.org

At this web site one can also get access to the Coq implementation of Iris.

Design Choices It is not obvious how one should introduce a sophisticated logical framework such as Iris, especially since Iris is a *framework* in more than one sense: Iris can be instantiated to reason about programs written in different programming languages and, moreover, Iris has a *base logic*, which can be used to define different kinds of program logics and relational models. We now describe some of the design choices we have made for these lecture notes.

These lecture notes are aimed at students with no prior knowledge of program logics. Hence we start from scratch and we focus on a particular instantiation of Iris to reason about a core concurrent higher-order imperative programming language, $\lambda_{\text{ref,conc}}$. (As Martin Hyland once put it [1]: "One good example is worth a host of generalities".)

We start with high-level concepts, such as Hoare triples and proof rules for those, and then, gradually, as we introduce more concepts, we show, *e.g.*, how proof rules that were postulated at first can be derived from simpler concepts. Moreover, new logical concepts are introduced with concrete, but often artificial, verification examples. The lecture notes also include larger case studies which show the logic can be used for verification of realistic programs.

Since the Iris logic involves several new logical modalities and connectives, we present example proofs of programs in a fairly detailed style (instead of the often-used proof outlines). We hope this will help readers learn how the novel aspects of the logic work.

We have included numerous exercises of varying degree of difficulty. Some exercises introduce reasoning principles used later in the notes. Thus the exercises are an integral part of the lecture notes, and should not be skipped.

When we introduce the logic, we only use intuitive semantics to explain why proof rules are sound. We defer a precise description of the semantics to a section which comes fairly late in the notes. There are several reasons for this choice: the formal semantics is non-trivial (*e.g.*, it involves solutions to recursive domain equations); the semantics is really defined for the base logic, which is only introduced later; and, finally, our experience from teaching a course based on the these lecture notes is that students can learn to use the logic without being exposed to the formal semantics of the logic.

Since Iris comes with a Coq implementation, it would perhaps be tempting to teach Iris using the Coq implementation from the beginning. However, we have decided against doing so. The reason is that our students do not have enough experience with Coq to make such an approach viable and, moreover, we believe that, for most readers, there would be too many things to learn at the same time. We do include a section on the Coq implementation and

also describe all the parts of Iris needed in order to work with the Coq implementation. The examples in the notes have been formalized in the Iris Coq implementation and are available at the Iris Project web site.

We have not attempted to include references to original research papers or to include historical remarks. Please see the Iris research papers for references to earlier work.

1 Introduction

The goal of these notes is to introduce a powerful logic, called Iris, for proving *partial* functional correctness of concurrent higher-order imperative programs. *Partial correctness* refers to the fact that when a program is proved correct with respect to some specification this only guarantees that *if the program terminates* then its result will satisfy the stated property. If a program does not terminate then the specification says nothing about its behaviour, apart from that it does not get stuck. (Knowing that an infinite computation does not get stuck can also be useful: it means that the program is safe and thus in particular that there are no memory errors such as trying to read from a location which does not exist in memory.)

Iris is a higher-order logic. This means in particular that program specifications can be parametrized by arbitrary propositions. One of the main benefits of this is that the generality of higher-order logic specifications support modularity: libraries and modules can be specified and proved correct once and for all, and different clients, or users, of the library can be verified in isolation, using only the specification (not the implementation) of the library the clients are using.

Iris supports verification of concurrent programs, by the use of so-called *ghost state*, and *invariants*. *Invariants* are a mechanism that allows different program threads to access shared resources, *e.g.*, read and write to the same location, provided they do not invalidate properties other threads depend on, *i.e.*, provided they maintain invariants. *Ghost state* is a mechanism which allows the invariants to evolve over time. It allows the logic to keep track of additional information that is not present in the program code being verified, but is nonetheless essential to proving its correctness, such as relationships between values of different program variables.

In thes notes, we introduce Iris gradually, starting with the necessary ingredients for reasoning about simple sequential programs, and refining and extending it until the logic is capable of reasoning about higher-order, concurrent, imperative programs. After that we show how the logic can be simplified into a minimal *base logic* in which all the rules which were used previously can be derived as theorems.

The examples used to introduce and explain the rules of the logic are often minimal and somewhat contrived. However the notes also contain larger *case studies*, in separate sections, which show that the logic can be used also to verify and reason about larger, and more realistic, programs. These case studies also illustrate modularity of the logic more clearly. More case studies can be found on the Iris project home page, iris-project.org.

In the last part of the notes we show consistency of the base logic by constructing a model and by showing soundness of the base logic with respect to the model.

2 Programming Language

A *program* logic is used to reason about *programs*, that is, to specify their behaviour. The precise rules of the logic depend on the constructs present in the programming language. Iris is really a framework, which can be instantiated to a range of different programming languages, but in order to learn about Iris, it is useful first to get some experience with one particular choice of programming language. Thus in this section we fix a concrete programming language, which we will use throughout the notes. Our language of choice, denoted $\lambda_{\text{ref,conc}}$, is an untyped higher-order ML-like language with general references and concurrency. Concurrency is supported via a primitive, fork $\{e\}$, for forking a new thread to compute e, and via a primitive compare and set, cas, operation. The syntax and the operational semantics is shown in Figure 1.

Syntax sugar In addition to the given constructs we will use additional syntax sugar when writing examples. We will write $\lambda x.e$ for the term $\mathsf{rec}\, f(x) = e$ where f is some fresh variable not appearing in e. Thus $\lambda x.e$ is a non-recursive function with argument x and body e. We will also write $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ for the term $(\lambda x.e_2)e_1$. This is the standard let expression, whose operational meaning (see the operational semantics below) is to evaluate the term e_1 to a value v, and then evaluate $e_2[v/x]$, *i.e.*, the term e_2 with the value v substituted for the variable x.

Operational semantics The operational semantics is defined by means of pure reductions, one-step reductions involving the heap, and general reductions among configurations. A configuation consists of a heap and a thread pool, and a thread pool is a mapping from thread identifiers (natural numbers) to expressions. Note that reduction of configuations is nondeterministic: we may choose to reduce in any thread $i \in \mathrm{dom}(\mathcal{E})$. Further note that reduction of a $\mathsf{fork}\, \{e\}$ expression in thread i proceeds by creating a new thread, with thread identifier j, whose initial expression is e, and that the result of evaluation $\mathsf{fork}\, \{e\}$ is the unit value $()$. Evaluation contexts are used to specify the evaluation strategy, that is, where the next reduction in a thread may take place. We use a call-by-value, left-to-right, evaluation strategy. Left-to-right refers to the evaluation of function applications, pairs, and binary operations, assignment, and compare and set cas. For example, the pair (e_1, e_2) is evaluated by first evaluating e_1 (the *left*most term), and then e_2. We finally remark on the compare and set cas primitive: in a heap h, compare and set $\mathsf{cas}(\ell, v, v')$ *atomically*, that is, *in one reduction step*, looks up the value of ℓ in h, compares it to v, and if it equals v, updates $h(\ell)$ to contain v'. (On real machines, cas may only be used to compare values that fit in a machine word, such as pointers and integers – for simplicity, we do not formalize such a restriction here; see [6] for an example of how it may be done.)

Let us illustrate an execution of a simple concurrent program. The program allocates a new location with initial value 0 and creates a new thread which updates the location to 3 and terminates. The main, *i.e.*, original, thread waits until the location has been changed from 0 at which point it reads it and adds 1 to it. The program is written as follows.

$$\begin{aligned}&\mathsf{let}\, x = \mathsf{ref}(0)\, \mathsf{in}\\ &\mathsf{let}\, y = \mathsf{fork}\, \{\mathsf{cas}(x, 0, 3)\}\, \mathsf{in}\\ &(\mathsf{rec}\, f() = \mathsf{if}\, !x = 0\, \mathsf{then}\, f()\, \mathsf{else}\, x \leftarrow\, !x + 1)()\end{aligned}$$

Let us call it e.

Exercise 2.1. Come up with at least two different executions of this program. ◊

3 The logic of resources

Iris is a higher-order separation logic over a simple type theory. Terms and types are those of simply typed lambda calculus extended with new base types and operations on them. These are provided by a signature $\mathcal{S}$. Note that this language is different from the *programming language* introduced in Section 2. It is unfortunate that the notation is often very similar, *e.g.*,, both the language of terms of Iris as well as the programming language have lambda abstraction, pairs, sums. We hope the reader will get used to the distinction.

We introduce the different notions of Iris step by step, starting with a minimal separation logic useful for a sequential language.

Syntax. Iris syntax is built up from a signature $\mathcal{S}$ and a countably infinite set *Var* of variables (ranged over by metavariables x, y, z):

Syntax

$$
\begin{array}{llll}
 & x, y, f & \in & \mathit{Var} \\
 & \ell & \in & \mathit{Loc} \\
 & n & \in & \mathbb{Z} \\
 & \circledcirc & ::= & + \mid - \mid * \mid = \mid < \mid \cdots \\
\mathit{Val} & v & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \ell \mid (v, v) \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_2\, v \mid \mathsf{rec}\, f(x) = e \\
\mathit{Exp} & e & ::= & x \mid n \mid e \circledcirc e \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \ell \\
 & & \mid & (e, e) \mid \pi_1\, e \mid \pi_2\, e \mid \mathsf{inj}_1\, e \mid \mathsf{inj}_2\, e \mid \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, y \Rightarrow e\, \mathsf{end} \\
 & & \mid & \mathsf{rec}\, f(x) = e \mid e\ e \\
 & & \mid & \mathsf{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid \mathsf{cas}(e, e, e) \mid \mathsf{fork}\, \{e\} \\
\mathit{ECtx} & E & ::= & - \mid E \circledcirc e \mid v \circledcirc E \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e \mid (E, e) \mid (v, E) \mid \pi_1\, E \mid \pi_2\, E \mid \mathsf{inj}_1\, E \mid \mathsf{inj}_2\, E \\
 & & \mid & \mathsf{match}\, E\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, y \Rightarrow e\, \mathsf{end} \mid E\, e \mid v\, E \mid \mathsf{ref}(E) \mid\ !\, E \mid E \leftarrow e \mid v \leftarrow E \\
 & & \mid & \mathsf{cas}(E, e, e') \mid \mathsf{cas}(v, E, e) \mid \mathsf{cas}(v, v', E) \\
\mathit{Heap} & h & \in & \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \\
\mathit{TPool} & \mathcal{E} & \in & \mathbb{N} \xrightarrow{\text{fin}} \mathit{Exp} \\
\mathit{Config} & \varsigma & ::= & (h, \mathcal{E})
\end{array}
$$

Pure reduction

$$
\begin{array}{rll}
v \circledcirc v' & \overset{\text{pure}}{\rightsquigarrow} v'' & \text{if } v'' = v \circledcirc v' \\
\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 & \overset{\text{pure}}{\rightsquigarrow} e_1 & \\
\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 & \overset{\text{pure}}{\rightsquigarrow} e_2 & \\
\pi_i\, (v_1, v_2) & \overset{\text{pure}}{\rightsquigarrow} v_i & \\
\mathsf{match}\, \mathsf{inj}_i\, v\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end} & \overset{\text{pure}}{\rightsquigarrow} e_i[v/x_i] & \\
(\mathsf{rec}\, f(x) = e)\, v & \overset{\text{pure}}{\rightsquigarrow} e[(\mathsf{rec}\, f(x) = e)/f, v/x] &
\end{array}
$$

Per-thread one-step reduction

$$
\begin{array}{rl}
(h, e) \rightsquigarrow (h, e') & \text{if } e \overset{\text{pure}}{\rightsquigarrow} e' \\
(h, \mathsf{ref}(v)) \rightsquigarrow (h[\ell \mapsto v], \ell) & \text{if } \ell \notin \mathrm{dom}(h) \\
(h, !\, \ell) \rightsquigarrow (h, h(\ell)) & \text{if } \ell \in \mathrm{dom}(h) \\
(h, \ell \leftarrow v) \rightsquigarrow (h[\ell \mapsto v], ()) & \text{if } \ell \in \mathrm{dom}(h) \\
(h, \mathsf{cas}(\ell, v_1, v_2)) \rightsquigarrow (h[\ell \mapsto v_2], \mathsf{true}) & \text{if } h(\ell) = v_1 \\
(h, \mathsf{cas}(\ell, v_1, v_2)) \rightsquigarrow (h, \mathsf{false}) & \text{if } h(\ell) \neq v_1
\end{array}
$$

Configuration reduction

$$
\frac{(h, e) \rightsquigarrow (h', e')}{(h, \mathcal{E}[i \mapsto E[e]]) \rightarrow (h', \mathcal{E}[i \mapsto E[e']])} \qquad \frac{j \notin \mathrm{dom}(\mathcal{E}) \cup \{i\}}{(h, \mathcal{E}[i \mapsto E[\mathsf{fork}\, \{e\}]]) \rightarrow (h, \mathcal{E}[i \mapsto E[()][j \mapsto e]])}
$$

Figure 1: Syntax and Operational Semantics of $\lambda_{\text{ref,conc}}$.

The types of Iris are built up from the following grammar, where T stands for additional base types which we will add later, *Val* and *Exp* are types of values and expressions in the language, and Prop is the type of Iris propositions.

$$\tau ::= T \mid \mathbb{Z} \mid \mathit{Val} \mid \mathit{Exp} \mid \mathsf{Prop} \mid 1 \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau$$

The corresponding terms of Iris are (recall that in higher-order logic propositions are also terms) defined below. They will be extended later when we introduce new concepts of Iris, and some of the terms that we treat as primitive now will turn out to be defined concepts.

$$\begin{aligned} t, P ::= {} & x \mid n \mid v \mid e \mid F(t_1, \ldots, t_n) \mid \\ & () \mid (t, t) \mid \pi_i\, t \mid \lambda x : \tau.\, t \mid t(t) \mid \\ & \mathsf{inl}\, t \mid \mathsf{inr}\, t \mid \mathsf{case}(t, x.t, y.t) \mid \\ & \mathsf{False} \mid \mathsf{True} \mid t =_\tau t \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid P * P \mid P \mathrel{-\!\!*} P \mid \\ & \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \\ & \Box P \mid \triangleright P \mid \\ & \{P\}\, t\, \{P\} \mid \\ & t \hookrightarrow t \end{aligned}$$

where x are variables, n are integers, v and e range over values of the language (*i.e.*, they are primitive terms of types *Val* and *Exp*), and F ranges over the function symbols in the signature $\mathcal{S}$.

The term () is the only term of the unit type 1, (t, t) are pairs, $\pi_i t$ is the projection, $\lambda x : \tau.\, t$ is the lambda abstraction and $t(t)$ denotes function application. Next there are introduction forms for sums (inl and inr) and the corresponding elimination form case.

The rest of the terms are logical constructs. Most of them are standard propositional connectives. The additional constructs are separating conjunction ($*$) and magic wand ($\mathrel{-\!\!*}$), which will be explained in Section 3. Then there are the later modality $\triangleright P$, explained in Section 5, and the always modality $\Box P$, explained in Section 6. Finally we have the Hoare triples $\{P\}\, t\, \{P\}$ and the points-to predicate $t \hookrightarrow t$, which are explained in Section 4.

The typing rules of the language of terms are shown in Figure 2. The judgments take the form $\Gamma \vdash_{\mathcal{S}} t : \tau$ and express when a term t has type τ in context Γ, given signature $\mathcal{S}$. The variable context Γ assigns types to variables of the logic. It is a list of pairs of a variable x and a type τ such that all the variables are distinct. We write contexts in the usual way, *e.g.*, $x_1 : \tau_1, x_2 : \tau_2$ is a context.

3.1 Propositions and entailment

The entailment rules of the logic are of the form

$$\Gamma \mid P \vdash Q$$

and, as usual, intuitively express that Q is provable from assumption P. Here P and Q are supposed to be well-typed propositions, *i.e.*, $\Gamma \vdash P : \mathsf{Prop}$ and $\Gamma \vdash Q : \mathsf{Prop}$.

When stating the rules of the logic we omit the context Γ if it does not change from premises to the conclusion of the rule. Moreover if there are multiple premises then we assume that the omitted context Γ is the same in all of them. The rules can be found in Figure 3. The first set

Well-typed terms of the basic logic $\boxed{\Gamma \vdash_{\mathcal{S}} t : \tau}$

$$\frac{}{x : \tau \vdash x : \tau} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma, x : \tau' \vdash t : \tau} \qquad \frac{\Gamma, x : \tau', y : \tau' \vdash t : \tau}{\Gamma, x : \tau' \vdash t[x/y] : \tau} \qquad \frac{\Gamma_1, x : \tau', y : \tau'', \Gamma_2 \vdash t : \tau}{\Gamma_1, x : \tau'', y : \tau', \Gamma_2 \vdash t[y/x, x/y] : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash t_n : \tau_n \qquad F : \tau_1, \ldots, \tau_n \to \tau_{n+1} \in \mathcal{F}}{\Gamma \vdash F(t_1, \ldots, t_n) : \tau_{n+1}} \qquad \frac{v \in \mathit{Val}}{\Gamma \vdash v : \mathit{Val}} \qquad \frac{e \in \mathit{Exp}}{\Gamma \vdash e : \mathit{Exp}}$$

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash u : \tau_2}{\Gamma \vdash (t, u) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, t : \tau_i} \qquad \frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x.\, t : \tau \to \tau'}$$

$$\frac{\Gamma \vdash t : \tau \to \tau' \qquad u : \tau}{\Gamma \vdash t(u) : \tau'}$$

$$\frac{}{\Gamma \vdash \mathsf{False} : \mathsf{Prop}} \qquad \frac{}{\Gamma \vdash \mathsf{True} : \mathsf{Prop}} \qquad \frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t =_\tau u : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \Rightarrow Q : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \wedge Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \vee Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P * Q : \mathsf{Prop}}$$

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \mathrel{-\!\!*} Q : \mathsf{Prop}} \qquad \frac{\Gamma, x : \tau \vdash P : \mathsf{Prop}}{\Gamma \vdash \exists x : \tau.\, P : \mathsf{Prop}} \qquad \frac{\Gamma, x : \tau \vdash P : \mathsf{Prop}}{\Gamma \vdash \forall x : \tau.\, P : \mathsf{Prop}}$$

Figure 2: Typing Rules for Terms of the Logic

of rules are the standard entailment rules of intuitionistic[1] higher-order logic. In addition, we have rules for the new logical connectives $*$ and $-\!\!*$. We explain the new rules in the following.

In Figure 4 on page 13 we list additional rules which state how different connectives interact. These rules are derivable. We show one example and leave the others, which are analogous, as exercises.

Example 3.1. We show

$$\frac{}{P * (Q \vee R) \dashv\vdash P * Q \vee P * R}.$$

The direction from right to left is immediate since we have $P*Q \vdash P*(Q\vee R)$ and $P*R \vdash P*(Q\vee R)$ by monotonicity of $*$ and $\vee$-introduction.

The direction from left to right relies on the existence of the wand. Indeed, for those familiar with adjoint functors, it is a consequence of the general categorical fact that left adjoints preserve colimits.

Consider the following proof tree (where we omit use of structural rules such as commutativity of $*$)

$$\dfrac{\dfrac{\dfrac{\dfrac{P*Q \vdash P*Q}{P*Q \vdash P*Q \vee P*R}}{Q \vdash P -\!\!* (P*Q \vee P*R)} \qquad \dfrac{\dfrac{P*R \vdash P*R}{P*R \vdash P*Q \vee P*R}}{R \vdash P -\!\!* (P*Q \vee P*R)}}{Q \vee R \vdash P -\!\!* (P*Q \vee P*R)}}{P*(Q\vee R) \vdash P*Q \vee P*R}.$$

Notice we made essential use of the following two rules

$$\frac{P * Q \vdash R}{P \vdash Q -\!\!* R} \qquad\qquad \frac{P \vdash Q -\!\!* R}{P * Q \vdash R}$$

The first is the introduction rule for $-\!\!*$, and the second one is easily derivable. We make use of the $-\!\!*$ to "move" the part of the context into the conclusion. This allows us to use the elimination rule for disjunction. ■

The other rules are derivable in an analogous way.

Exercise 3.2. Following the example above derive the following two rules:

$$\frac{x \notin \mathrm{FV}(P)}{P * \exists x.\, \Phi \dashv\vdash \exists x.\, P * \Phi} \qquad\qquad \frac{x \notin \mathrm{FV}(P)}{P \wedge \exists x.\, \Phi \dashv\vdash \exists x.\, P \wedge \Phi}$$

◊

Terminology A word about terminology. We generally use "proposition" for terms of type Prop and "predicate" for terms of type $\tau \to \mathsf{Prop}$ for types τ. However the distinction is not so clear since, if $P : \tau \to \mathsf{Prop}$ is a predicate and $x : \tau$, then Px is a proposition. Moreover, propositions can be thought of as nullary predicates, that is, predicates on the type 1. Thus the decision of when to use which term is largely a matter of convention and is not significant.

[1] Intuitionistic refers to the fact that we *do not* assume the law of excluded middle $\mathsf{True} \vdash P \vee \neg P$. This law is incompatible with some of the features of the logic we introduce later.

Intuition for Iris propositions Intuitively, an Iris proposition describes a set of resources. A canonical example of a resource is a heap fragment. So far we have not introduced any primitives for talking about resources. One such primitive is the "points-to" predicate $x \hookrightarrow v$, which we will use extensively in Section 4 in connection with Hoare triples. For now it is enough to think of $x \hookrightarrow v$ as follows. It describes the set of all heap fragments that map location x to value v (in particular location x needs to exist in the heap).

In addition there is another reading, another intuition, for Iris propositions. Propositions *assert ownership of resources*. For example, $x \hookrightarrow v$ asserts that we have the sole authority, *i.e.*, exclusive ownership, of the portion of the heap which contains the location x. This means that we are free to read and modify the value stored at x. This intuition will become clearer in connection with Hoare triples, and the ownership reading of propositions will become particularly useful when programs contain multiple threads.

With this intuition the proposition $P * Q$ describes the set of resources which can be split up into two disjoint parts, with one part described by P and the other part described by Q.

For example, $x \hookrightarrow u * y \hookrightarrow v$ describes the set of heaps with two *disjoint* locations x and y, the first stores u and the second v.

The proposition $P \mathrel{-\!\!*} Q$ describes those resources r which satisfy that, if we combine r with a disjoint resource described r' by P, then we get a resource described by Q. For example, the proposition

$$x \hookrightarrow u \mathrel{-\!\!*} (x \hookrightarrow u * y \hookrightarrow v)$$

describes those heap fragments that map y to v, because when we combine it with a heap fragment mapping x to u, then we get a heap fragment mapping x to u and y to v.

In the following section it suffices to think of resources as heap fragments. Later on, we will see much more sophisticated notions of resource and much more refined notions of ownership, including shared ownership, than that captured by simple points-to predicate. When r is a resource described by P, we also say that r *satisfies* P or that r *is in* P.

Entailment relation Entailment rules are often presented using a sequence of formulas together with a number of structural rules, which manipulate such sequences of assumptions. Here instead, the assumption of the entailment rules consists of only a single formula, and the structural rules are replaced by appropriate uses of transitivity of entailment together with properties of conjunction and separating conjunction, such as associativity, commutativity, and weakening. We have chosen this single-assumption style of presentation because otherwise we would have needed two ways of extending the sequence of formulas, one corresponding to ordinary conjunction and one corresponding to separating conjunction. The intuition reading of an entailment $P \vdash Q$ is that, for all resources r, if r is in P, then r is also in Q.

3.2 Rules for separating conjunction and magic wand

We now explain the logical entailments for separating conjunction and the magic wand based on the resource reading of propositions. First recall that we read the proposition $P * Q$ as containing those resources r which can be split into two resources r_1 and r_2, with r_1 satisfying P and r_2 satisfying Q. What it means to split a resource is dependent on the particular notion of a resource. For example, if the resources are heap fragments then splitting means splitting the heap: some locations go to one subheap, the others to the other.

Weakening The rule

$$\frac{}{P_1 * P_2 \vdash P_1}\ *\text{-WEAK}$$

states that we can forget about resources. This makes Iris an *affine* separation logic.[2] With the resource reading of propositions this rule restricts the kind of sets that can be allowed as sets of resources.

For instance, if resources are heaps then the points-to predicate $x \hookrightarrow v$ contains those heaps which map x to v, but other locations can contain values as well. Then, for instance, the rule

$$x \hookrightarrow u * y \hookrightarrow v \vdash x \hookrightarrow u$$

is clear: On the left-hand side we have heaps which map the location x to u and the location y to v. Any such heap in particular maps x to u, so is in the set of resources described by the right-hand side.

The associativity and commutativity rules

$$\frac{}{P_1 * (P_2 * P_3) \dashv\vdash (P_1 * P_2) * P_3}\ *\text{-ASSOC} \qquad \frac{}{P_1 * P_2 \dashv\vdash P_2 * P_1}\ *\text{-COMM}$$

are basic structural rules. The symbol $\dashv\vdash$ is used to indicate that the rule can be used in both directions (so that we do not have to write two separate rules for associativity). The above rules hold because "to separate" is a commutative and associative operation.

Separating conjunction introduction The introduction rule

$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}\ *\text{I}$$

states that to prove a separating conjunction $Q_1 * Q_2$ we need to split the assumptions as well and decide which ones to use to prove Q_1 and which ones to use to prove Q_2. Compared to the introduction rule for ordinary conjunction ($\wedge$I), this splitting of assumptions restricts the basic structural properties of $*$. For instance, $P \vdash P * P$ is not provable in general. However, this "limitation" allows us to state additional properties in combination with primitive resource assertions. For instance, if resources are heaps then we have the following basic property of the points-to predicate

$$x \hookrightarrow v * x \hookrightarrow u \vdash \mathsf{False}.$$

Note that if we used ordinary conjunction in the above axiom, then we would be able to derive $\neg(x \hookrightarrow v)$ for all x and v, making the points-to predicate useless.

Magic wand introduction and elimination

$$\frac{R * P \vdash Q}{R \vdash P \mathrel{-\!\!*} Q}\ {-\!\!*}\text{I} \qquad \frac{R_1 \vdash P \mathrel{-\!\!*} Q \qquad R_2 \vdash P}{R_1 * R_2 \vdash Q}\ {-\!\!*}\text{E}$$

[2] Sometimes called intuitionistic separation logic, but that terminology is ambiguous since it can also refer to the absence of the law of excluded middle or other classical axioms.

The magic wand $P \mathrel{-\!\!*} Q$ is akin to the difference of resources in Q and those in P: it is the set of all those resources which when combined with any resource in P are in Q. With this intuition the introduction rule should be intuitively clear.

The elimination rule is similar to the elimination rule for implication ($\Rightarrow$E), except that we need to split the assumptions and decide which ones to use to prove the magic wand and which ones to use to prove the premise of the magic wand.

3.3 Basic mathematical constructions in Iris

Reasoning about programs involves translating effectful behaviour into mathematical specifications, *e.g.*, a program manipulating linked lists will be specified by using operations on mathematical sequences. For this reason we need to define and reason about such mathematical objects in the logic. For the purposes of these lecture notes we will assume that we can define and reason about such objects as in ordinary mathematics. This is justified since it is known that in a higher-order logic with a type of natural numbers and suitable induction and recursion principles for it, most mathematical concepts, such as lists, trees, integers, arithmetic, rational, real, and complex numbers, are definable. The encoding is beyond the scope of this note, so we omit it.

4 Separation logic for sequential programs

To get basic separation logic we extend the basic logic of resources with two new predicates: Hoare triples and the points-to predicate. Hoare triples are basic predicates about programs (terms of type *Exp*) and the points-to predicate $x \hookrightarrow v$ is a basic proposition about resources, which at this stage can be thought of as heap fragments.

The new constructs satisfy the following typing rules.

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash e : \mathit{Exp} \qquad \Gamma \vdash \Phi : \mathit{Val} \to \mathsf{Prop}}{\Gamma \vdash \{P\}\, e\, \{\Phi\} : \mathsf{Prop}} \qquad \frac{\Gamma \vdash \ell : \mathit{Val} \qquad \Gamma \vdash v : \mathit{Val}}{\Gamma \vdash \ell \hookrightarrow v : \mathsf{Prop}}$$

The intuitive reading of the Hoare triple $\{P\}\, e\, \{\Phi\}$ is that if the program e is run in a heap h nsatisfying P, then the computation does not get stuck and, moreover, if it terminates with a value v and a heap h', then h' satisfies $\Phi(v)$. Note that Φ has two purposes. It describes the value v, *e.g.*, it could contain the proposition $v = 3$, and it describes the resources after the execution of the program, *e.g.*, it could contain $x \hookrightarrow 15$. At this stage the precondition P must describe the set of resources necessary for e to run safely, if we are to prove the triple. Informally, we sometimes say that P must include the *footprint* of e, those resources needed for e to run safely. For example, if the expression e uses a location during evaluation (either reads to or writes from it), then any heap fragment in P must contain a value at that location. As a consequence, if the expression is run in a heap with location ℓ, which is not mentioned by P, then the value at that location will not be altered. This is one intuition for why the frame rule below is sound.

Later on, in Section 7, not all resources needed to execute e will need to be in the precondition. Resources shared by different threads will instead be in *invariants*, and only resources that are *local* to the thread will be in preconditions. But that is for later.

Laws for the points-to predicate As we described above the basic predicate $x \hookrightarrow v$ describes those heap fragments that map the location x to the value v.

We have the usual η and β laws for projections, λ and μ.

$$\textsc{unit-}\eta\ \frac{\Gamma \vdash t : 1}{\Gamma \vdash t \equiv ()} \qquad \lambda-\beta\ \frac{}{\Gamma \vdash (\lambda x : \tau.\, e_1)(e_2) \equiv e_1[e_2/x]} \qquad \lambda-\eta\ \frac{x \text{ not free in } e \qquad \Gamma \vdash e : \tau_1 \to \tau_2}{\Gamma \vdash (\lambda x.e(x)) \equiv e}$$

$$\pi-\beta\ \frac{}{\Gamma \vdash \pi_i(e_1, e_2) \equiv e_i} \qquad \pi-\eta\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash e \equiv (\pi_1 e, \pi_2 e)} \qquad \text{inl-}\beta\ \frac{}{\Gamma \vdash \mathsf{case}(\mathsf{inl}\, e, x.e_1, y.e_2) \equiv e_1[e/x]}$$

$$\text{inr-}\beta\ \frac{}{\Gamma \vdash \mathsf{case}(\mathsf{inr}\, e, x.e_1, y.e_2) \equiv e_2[e/y]} \qquad \text{case-}\eta\ \frac{\Gamma \vdash e : \tau + \sigma \qquad \Gamma, z : \tau + \sigma \vdash e_1 : \rho}{\Gamma \vdash \mathsf{case}(e, x.e_1[\mathsf{inl}\, x/z], y.e_1[\mathsf{inr}\, y/z]) \equiv e}$$

$$\mu-\beta\ \frac{}{\Gamma \vdash \mu x : \tau.\, e \equiv e[\mu x : \tau.\, e/x]}$$

Laws of intuitionistic higher-order logic with equality. Standard rules.

$$\textsc{Asm}\ \frac{}{P \vdash P} \qquad \textsc{Trans}\ \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R} \qquad \textsc{Eq}\ \frac{\Gamma, x : \tau \vdash Q : \mathsf{Prop} \qquad \Gamma \mid P \vdash Q[t/x] \qquad \Gamma \mid P \vdash t =_\tau t'}{\Gamma \mid P \vdash Q[t'/x]} \qquad \textsc{Refl}\ P \vdash t =_\tau t$$

$$\bot\text{E}\ \frac{Q \vdash \mathsf{False}}{Q \vdash P} \qquad \top\text{I}\ \frac{}{Q \vdash \mathsf{True}} \qquad \wedge\text{I}\ \frac{R \vdash P \qquad R \vdash Q}{R \vdash P \wedge Q} \qquad \wedge\text{EL}\ \frac{R \vdash P \wedge Q}{R \vdash P} \qquad \wedge\text{ER}\ \frac{R \vdash P \wedge Q}{R \vdash Q} \qquad \vee\text{IL}\ \frac{R \vdash P}{R \vdash P \vee Q}$$

$$\vee\text{IR}\ \frac{R \vdash Q}{R \vdash P \vee Q} \qquad \vee\text{E}\ \frac{S \vdash P \vee Q \qquad S \wedge P \vdash R \qquad S \wedge Q \vdash R}{S \vdash R} \qquad \Rightarrow\text{I}\ \frac{R \wedge P \vdash Q}{R \vdash P \Rightarrow Q} \qquad \Rightarrow\text{E}\ \frac{R \vdash P \Rightarrow Q \qquad R \vdash P}{R \vdash Q}$$

$$\forall\text{I}\ \frac{\Gamma, x : \tau \mid Q \vdash P}{\Gamma \mid Q \vdash \forall x : \tau.\, P} \qquad \forall\text{E}\ \frac{\Gamma \mid Q \vdash \forall x : \tau.\, P \qquad \Gamma \vdash t : \tau}{\Gamma \mid Q \vdash P[t/x]} \qquad \exists\text{I}\ \frac{\Gamma \mid Q \vdash P[t/x] \qquad \Gamma \vdash t : \tau}{\Gamma \mid Q \vdash \exists x : \tau.P}$$

$$\exists\text{E}\ \frac{\Gamma \mid R \vdash \exists x : \tau.\, P \qquad \Gamma, x : \tau \mid R \wedge P \vdash Q}{\Gamma \mid R \vdash Q}$$

Laws of (affine) bunched implications.

$$*\text{-WEAK}\ \frac{}{P_1 * P_2 \vdash P_1} \qquad *\text{-ASSOC}\ \frac{}{P_1 * (P_2 * P_3) \dashv\vdash (P_1 * P_2) * P_3} \qquad *\text{-COMM}\ \frac{}{P_1 * P_2 \dashv\vdash P_2 * P_1} \qquad *\text{I}\ \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}$$

$$-\!\!*\text{I}\ \frac{R * P \vdash Q}{R \vdash P -\!\!* Q} \qquad -\!\!*\text{E}\ \frac{R_1 \vdash P -\!\!* Q \qquad R_2 \vdash P}{R_1 * R_2 \vdash Q}$$

Figure 3: Logical entailment

$$\frac{}{P * (Q \vee R) \dashv\vdash P * Q \vee P * R} \qquad \frac{x \notin P}{P * \exists x.\, \Phi \dashv\vdash \exists x.\, P * \Phi} \qquad \frac{x \notin P}{P \wedge \exists x.\, \Phi \dashv\vdash \exists x.\, P \wedge \Phi}$$

Figure 4: Derivable rules for interaction of connectives.

The essential properties of the points-to predicate are that (1) it is *not* duplicable, which means that

$$\ell \hookrightarrow v * \ell \hookrightarrow v' \vdash \mathsf{False}$$

and (2) that it is a partial function in the sense that

$$\ell \hookrightarrow v \wedge \ell \hookrightarrow v' \vdash v =_{Val} v'.$$

Other properties of the points-to predicate come into play in connection with Hoare triples and language constructs which manipulate state.

Laws for Hoare triples The basic axioms for Hoare triples are listed in Figure 5 on page 16. They are split into three groups. First there are structural rules. These deal with transforming pre- and postconditions but do not change the program. Next, for each elimination form and basic heap operation of the language, there is a rule stating how the primitive operation transform pre- and postconditions. In the third group we list two more structural rules which allow us to move persistent propositions, that is, propositions which do not depend on any resources, in and out of preconditions.

In the postconditions, we use the notation $v.Q$ to mean $\lambda v.Q$.

In most rules there is an arbitrary proposition/assumption S. Some structural rules, such as Ht-Eq do change it, but in most rules it remains unchanged. This proposition is necessary. It will contain, *e.g.*, equalities or inequalities between natural numbers, and other facts about terms appearing in triples. We now explain the rules.

The frame rule The frame rule

$$\textsc{Ht-frame}\ \frac{S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P * R\}\, e\, \{v.Q * R\}}$$

expresses that if an expression e satisfies a Hoare triple, then it will preserve any resources described by a "frame" R (*i.e.*, environment resources not touched by the evaluation of the term) R disjoint from the resources described by the precondition P. Intuitively, this frame rule is sound since the precondition P of a Hoare triple $\{P\}\, e\, \{v.Q\}$ describes the whole footprint of e, which is to say that e does not touch any other resources disjoint from those in P.

For example, a method that swaps the values stored at two locations can be specified by only mentioning those two locations. Then, when the specification is used, there will typically be a frame R, asserting facts about other resources. Since the swap function does not rely on them nor does it alter them, the frame R will be preserved.

In original Hoare logic without separating conjunction such a rule was not expressible. Note that the use of separating conjunction as opposed to conjunction is essential. The rule

$$\textsc{Ht-frame-invalid}\ \frac{S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P \wedge R\}\, e\, \{v.Q \wedge R\}}$$

is not sound.

Exercise 4.1. Come up with a counterexample to the above rule. ◊

Exercise 4.2. Prove

$$S \vdash \{P\}\, e\, \{v.Q\} \Rightarrow \forall R : \mathsf{Prop}, \{P * R\}\, e\, \{v.Q * R\}$$

◊

False precondition The following rule

$$\textsc{Ht-False}\ \frac{}{S \vdash \{\mathsf{False}\}\, e\, \{v.Q\}}$$

is trivially sound since there are no resources satisfying False.

Value and evaluation context rules

$$\textsc{Ht-ret}\ \frac{w \text{ is a value}}{S \vdash \{\mathsf{True}\}\, w\, \{v.v = w\}} \qquad \textsc{Ht-bind}\ \frac{E \text{ is an eval. context} \qquad S \vdash \{P\}\, e\, \{v.Q\} \qquad S \vdash \forall v.\, \{Q\}\, E[v]\, \{w.R\}}{S \vdash \{P\}\, E[e]\, \{w.R\}}$$

The Ht-ret rule is simple: Since a computation consisting of a value does not compute further, it does not require any resources and the return value is just the value itself.

The rule Ht-bind is more interesting and will be used extensively in order to transform the verification of a big program $E[e]$ to the verification of individual steps for which we have basic axioms for Hoare triples. To illustrate consider the following example.

Suppose we are to prove (using let expressions, which are definable in the language)

$$\{P\}\, \mathsf{let}\, x = e\, \mathsf{in}\, e_2\, \{v.R\}$$

and suppose we have already proved

$$\{P\}\, e\, \{v.Q\}$$

for some Q. The rule Ht-bind states that we only need to verify

$$\{Q[u/v]\}\, \mathsf{let}\, x = u\, \mathsf{in}\, e_2\, \{v.R\}$$

for all values u. Typically, Q will restrict the set of values; it will be those values which are possible results of evaluating e.

Exercise 4.3. Use Ht-bind to show $\{\mathsf{True}\}\, 3 + 4 + 5\, \{v.v = 12\}$. ◊

Persistent propositions Some of the propositions, namely Hoare triples and equality, do not rely on any resources, *i.e.*, they either hold for all resources, or none. We call such propositions *persistent* and we will see more examples, and a more uniform treatment, later on. For now the essential properties of persistent propositions are the rules HT-EQ and HT-HT, together with the following axiom for persistent propositions P

$$P \wedge Q \vdash P * Q \qquad \text{if } P \text{ is persistent.}$$

Intuitively, if P is persistent, then it does not depend on any resources. Hence if $P \wedge Q$ holds, then we can split any resource r in $P \wedge Q$ into an "empty resource" (*e.g.*, empty heap) and r. Then r belongs to Q and since some resource (namely r) belongs to P, so does the empty resource, because P is independent of resources. Later on this intuition will be slightly refined and persistent propositions will be allowed to depend on a certain restricted set of resources.

Note that we always have the entailment

$$P * Q \vdash P \wedge Q$$

and thus, if one of the propositions is *persistent*, then there is no difference between conjunction and separating conjunction.

Exercise 4.4. Prove the derived rule $P * Q \vdash P \wedge Q$. ◊

The rule of consequence

$$\textsc{Ht-csq}\quad \frac{S \text{ persistent} \qquad S \vdash P \Rightarrow P' \qquad S \vdash \{P'\}\, e\, \{v.\, Q'\} \qquad S \vdash \forall u.\, Q'[u/v] \Rightarrow Q[u/v]}{S \vdash \{P\}\, e\, \{v.\, Q\}}$$

The rule of consequence states that we can strengthen the precondition and weaken the post-condition. It is important that the context in which strengthening and weakening occur is persistent, that is, that it does not rely on any resources (hence the context must not contain predicates such as the points-to predicate). The rule of consequence is used very often, most of the time implicitly.

Elimination of disjunction and existential quantification The rules

$$\textsc{Ht-disj}\quad \frac{S \vdash \{P\}\, e\, \{v.\, R\} \qquad S \vdash \{Q\}\, e\, \{v.\, R\}}{S \vdash \{P \vee Q\}\, e\, \{v.\, R\}} \qquad\qquad \textsc{Ht-exist}\quad \frac{x \notin Q \qquad S \vdash \forall x.\, \{P\}\, e\, \{v.\, Q\}}{x \notin Q \qquad S \vdash \{\exists x.\, P\}\, e\, \{v.\, Q\}}$$

allow us to make use of disjunction and existential quantification in the precondition.

Exercise 4.5. Use the intuitive reading of Hoare triples to explain why the above rules are sound (note that both of them are double rules and remember to consider both directions). ◊

Rules for basic constructs of the language. The first rule appearing is the rule HT-OP internalising the operational semantics of binary operations.

Next is the rule for reading a memory location.

$$\textsc{Ht-load}\quad \frac{}{S \vdash \{\ell \hookrightarrow u\}\, !\ell\, \{v.v = u \wedge \ell \hookrightarrow u\}}$$

Structural rules.

HT-FRAME
$$\frac{S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P * R\}\, e\, \{v.Q * R\}}$$

HT-FALSE
$$\frac{}{S \vdash \{\mathsf{False}\}\, e\, \{v.Q\}}$$

HT-RET
$$\frac{w \text{ is a value}}{S \vdash \{\mathsf{True}\}\, w\, \{v.v = w\}}$$

HT-BIND
$$\frac{E \text{ is an eval. context} \qquad S \vdash \{P\}\, e\, \{v.Q\} \qquad S \vdash \forall v.\, \{Q\}\, E[v]\, \{w.R\}}{S \vdash \{P\}\, E[e]\, \{w.R\}}$$

HT-CSQ
$$\frac{S \text{ persistent} \qquad S \vdash P \Rrightarrow P' \qquad S \vdash \{P'\}\, e\, \{v.Q'\} \qquad S \vdash \forall u.\, Q'[u/v] \Rrightarrow Q[u/v]}{S \vdash \{P\}\, e\, \{v.Q\}}$$

HT-DISJ
$$\frac{S \vdash \{P\}\, e\, \{v.R\} \qquad S \vdash \{Q\}\, e\, \{v.R\}}{S \vdash \{P \vee Q\}\, e\, \{v.R\}}$$

HT-EXIST
$$\frac{x \notin Q \qquad S \vdash \forall x.\, \{P\}\, e\, \{v.Q\}}{x \notin Q \qquad S \vdash \{\exists x.\, P\}\, e\, \{v.Q\}}$$

Rules for basic constructs of the language.

HT-OP
$$\frac{v'' = v \circledcirc v'}{\{\mathsf{True}\}\, v \circledcirc v'\, \{r.r = v''\}}$$

HT-LOAD
$$\frac{}{S \vdash \{\ell \hookrightarrow u\}\, !\ell\, \{v.v = u \wedge \ell \hookrightarrow u\}}$$

HT-ALLOC
$$\frac{}{S \vdash \{\mathsf{True}\}\, \mathsf{ref}(u)\, \{v.\exists \ell.\, v = \ell \wedge \ell \hookrightarrow u\}}$$

HT-STORE
$$\frac{}{S \vdash \{\ell \hookrightarrow -\}\, \ell \leftarrow w\, \{v.v = () \wedge \ell \hookrightarrow w\}}$$

HT-REC
$$\frac{\Gamma, g : \mathit{Val} \mid S \wedge \forall y.\, \forall v.\, \{P\}\, g v\, \{u.Q\} \vdash \forall y.\, \forall v.\, \{P\}\, e[g/f, v/x]\, \{u.Q\}}{\Gamma \mid S \vdash \forall y.\, \forall v.\, \{P\}\, (\mathsf{rec}\, f(x) = e) v\, \{u.Q\}}$$

HT-PROJ
$$\frac{}{S \vdash \{\mathsf{True}\}\, \pi_i\, (v_1, v_2)\, \{v.v = v_i\}}$$

MATCH
$$\frac{S \vdash \{P\}\, e_i\, [u/x_i]\, \{v.Q\}}{S \vdash \{P\}\, \mathsf{match}\, \mathsf{inj}_i\, u\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end}\, \{v.Q\}}$$

HT-IF
$$\frac{\{P * v = \mathsf{true}\}\, e_2\, \{u.Q\} \qquad \{P * v = \mathsf{false}\}\, e_3\, \{u.Q\}}{\{P\}\, \mathsf{if}\, v\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3\, \{u.Q\}}$$

The following two rules allow us to move persistent propositions in and out of preconditions.

HT-EQ
$$\frac{S \wedge t =_\tau t' \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P \wedge t =_\tau t'\}\, e\, \{v.Q\}}$$

HT-HT
$$\frac{S \wedge \{P_1\}\, e_1\, \{v.Q_1\} \vdash \{P_2\}\, e_2\, \{v.Q_2\}}{S \vdash \{P_2 \wedge \{P_1\}\, e_1\, \{v.Q_1\}\}\, e_2\, \{v.Q_2\}}$$

Figure 5: Rules for Hoare triples.

To read we need resources: we need to know that the location ℓ we wish to read from exists and, moreover, that a value u is stored at that location. After reading we get the exact value and we still have the resources we started with. Note that it is crucial that the postcondition still contains $\ell \hookrightarrow u$. Otherwise it would be impossible to use the same location more than once if we wished to verify a program.

Allocation does not require any resources, *i.e.*, it is safe to run $\mathsf{ref}(u)$ in any heap. Hence the precondition for allocation is True.

HT-ALLOC

$$\frac{}{S \vdash \{\mathsf{True}\}\,\mathsf{ref}(u)\,\{v.\exists \ell.\, v = \ell \wedge \ell \hookrightarrow u\}}$$

We get back an $\ell \hookrightarrow u$ resource. That is, after executing $\mathsf{ref}(u)$ we know there exists some location ℓ which contains u. Note that we cannot choose which location we get – the exact location will depend on which locations are already allocated in the heap in which we run $\mathsf{ref}(u)$ – hence the existential quantification in the postcondition.

Writing to location ℓ

HT-STORE

$$\frac{}{S \vdash \{\ell \hookrightarrow -\}\,\ell \leftarrow w\,\{v.v = () \wedge \ell \hookrightarrow w\}}$$

requires resources. Namely that the location exists in the heap ($\ell \hookrightarrow -$ is shorthand for $\exists u.\ell \hookrightarrow u$). Note that with the ownership reading of Iris propositions the requirement that ℓ points-to some value means that we *own* the location. Hence we can change the value stored at it (without violating assumptions of other modules or concurrently running threads).

Remark 4.6. Note that it is essential that $\ell \hookrightarrow -$ is in the precondition of the HT-STORE, even though we do not care what is stored at the location. A rule such as

$$\frac{}{S \vdash \{\mathsf{True}\}\,\ell \leftarrow w\,\{v.v = () \wedge \ell \hookrightarrow w\}}$$

would lead to inconsistency together with the frame rule.

A conceptual reason why such a rule is inconsistent is that the resources required by the program to run should be in the precondition. This ensures that the program is safe to run, *i.e.*, it will not get stuck. And since the program will get stuck storing a value to a location if the location is not allocated, we should not be able to give it a specification that allows it to be run in such a heap, *i.e.*, with precondition True. ■

The rule for the conditional expression HT-IF is as expected.

HT-IF

$$\frac{\{P * v = \mathsf{true}\}\,e_2\,\{u.Q\} \qquad \{P * v = \mathsf{false}\}\,e_3\,\{u.Q\}}{\{P\}\,\mathsf{if}\,v\,\mathsf{then}\,e_2\,\mathsf{else}\,e_3\,\{u.Q\}}$$

It mimics the intuitive meaning of the conditional expression. The rules for eliminating values of the product and sum types

PROJ

$$\frac{}{S \vdash \{\mathsf{True}\}\,\pi_i\,(v_1, v_2)\,\{v.v = v_i\}}$$

MATCH

$$\frac{S \vdash \{P\}\,e_i\,[u/x_i]\,\{v.Q\}}{S \vdash \{P\}\,\mathsf{match}\,\mathsf{inj}_i\,u\,\mathsf{with}\,x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2\,\mathsf{end}\,\{v.Q\}}$$

are straightforward from the operational semantics of the language.

Finally, the recursion rule is interesting.

$$\textsc{Ht-Rec}\quad \frac{\Gamma, g : \mathit{Val} \mid S \wedge \forall y.\, \forall v.\, \{P\}\, g\, v\, \{u.Q\} \vdash \forall y.\, \forall v.\, \{P\}\, e[g/f, v/x]\, \{u.Q\}}{\Gamma \mid S \vdash \forall y.\, \forall v.\, \{P\}\, (\mathsf{rec}\, f(x) = e) v\, \{u.Q\}}$$

It states that to prove that the recursive function application satisfies some specification it suffices to prove that the body of the recursive function satisfies the specification *under the assumption* that all recursive calls satisfy it. The variable v is the programming language value to which the function is applied. The variable y is the logical variable, which will typically be connected to the programming language value v in the precondition P. As an example, which we will see in detail later on, the value v could be a pointer to a linked list, and the variable y could be the list of values stored in the linked list. What type precisely y ranges over depends on the precise application in mind. In some examples we will not need the variable y, *i.e.*, we shall use the rule

$$\frac{\Gamma, g : \mathit{Val} \mid S \wedge \forall v.\, \{P\}\, g\, v\, \{u.Q\} \vdash \forall v.\, \{P\}\, e[g/f, v/x]\, \{u.Q\}}{\Gamma \mid S \vdash \forall v.\, \{P\}\, (\mathsf{rec}\, f(x) = e) v\, \{u.Q\}} \tag{1}$$

This rule is derivable from the rule Ht-Rec by choosing the variable y to range over the singleton type 1 using the logical equivalence $\forall y : 1.\, \Phi \iff \Phi$, provided y does not appear in Φ.

Example 4.7. The recursion rule perhaps looks somewhat intimidating. To illustrate how it is used we use it to verify a very simple program, the program computing the factorial. The factorial function can be implemented in our language as follows.

$$\mathsf{rec}\, \mathsf{fac}(n) = \mathsf{if}\, n = 0\, \mathsf{then}\, 1\, \mathsf{else}\, n * \mathsf{fac}(n-1).$$

The specification we wish to give it is, of course,

$$\forall n. \{n \geq 0\}\, \mathsf{fac}\, n\, \{v.\, v =_{\mathit{Val}} n!\}$$

where $n!$ is the factorial of the number n.

Let us now sketch a proof of it. There is no logical variable y, so we will use the simplified rule (1). Using the rule we see that we must show the entailment (the context S is empty)

$$\forall n. \{n \geq 0\}\, f\, n\, \{v.\, v =_{\mathit{Val}} n!\} \vdash \forall n. \{n \geq 0\}\, \mathsf{if}\, n = 0\, \mathsf{then}\, 1\, \mathsf{else}\, n * f(n-1)\, \{v.\, v =_{\mathit{Val}} n!\}$$

So let us assume

$$\forall n. \{n \geq 0\}\, f\, n\, \{v.\, v =_{\mathit{Val}} n!\}. \tag{2}$$

To prove $\forall n. \{n \geq 0\}\, \mathsf{if}\, n = 0\, \mathsf{then}\, 1\, \mathsf{else}\, n * f(n-1)\, \{v.\, v =_{\mathit{Val}} n!\}$ we start by using the Ht-If rule. Thus we have to prove two triples

$$\{n \geq 0 * (n = 0) =_{\mathit{Val}} \mathsf{true}\}\, 1\, \{v.\, v =_{\mathit{Val}} n!\}$$
$$\{n \geq 0 * (n = 0) =_{\mathit{Val}} \mathsf{false}\}\, n * f(n-1)\, \{v.\, v =_{\mathit{Val}} n!\}$$

We leave the first one as an exercise and focus on the second. Using the Ht-bind with the evaluation context being $n * -$ and the intermediate assertion Q being $Q \equiv v = (n-1)!$ we have to prove the following two triples

$$\{n \geq 0 * (n = 0) =_{\mathit{Val}} \mathsf{false}\}\, f(n-1)\, \{v.v =_{\mathit{Val}} (n-1)!\}$$
$$\forall v. \{v = (n-1)!\}\, n * v\, \{u.u =_{\mathit{Val}} n!\}$$

The first triple follows by the rule of consequence and the assumption (2). Indeed $n \geq 0 * (n = 0) =_{Val} \mathsf{false}$ implies $n - 1 \geq 0$, and instantiating the assumption (2) with $n - 1$ we get

$$\{n - 1 \geq 0\}\, f\,(n-1)\,\{v.\, v =_{Val} (n-1)!\}$$

as needed. The second triple follows easily by Ht-Eq and basic properties of equality and the factorial function. ■

Notice that rules above are stated in very basic form with values wherever possible, which means they are often needlessly cumbersome to use in their basic form. The following exercises develop some derived rules.

Exercise 4.8. Prove the following derived rule. For any *value* u and expression e we have

$$\frac{S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P\}\, \pi_1(e, u)\, \{v.Q\}}.$$

It is important that the second component is a value. Show that the following rule is not valid in general if e_1 is allowed to be an arbitrary expression.

$$\frac{S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P\}\, \pi_1(e, e_1)\, \{v.Q\}}.$$

Hint: What if e and e_1 read or write to the same location?

The problem is that we know nothing about the behaviour of e_1. But we can specify its behaviour using Hoare triples. Come up with some propositions P_1 and P_2 or some conditions on P_1 and P_2 such that the following rule

$$\frac{S \vdash \{P\}\, e\, \{v.Q\} \qquad S \vdash \{P_1\}\, e_1\, \{v.P_2\}}{S \vdash \{P\}\, \pi_1(e, e_1)\, \{v.Q\}}.$$

is derivable. ◊

Exercise 4.9. From Ht-If we can derive two, perhaps more natural, rules, which are simpler to use. They require us to only prove a specification of the branch which will be taken.

Ht-If-True
$$\frac{\{P * v = \mathsf{true}\}\, e_2\, \{u.Q\}}{\{P * v = \mathsf{true}\}\, \mathsf{if}\, v\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3\, \{u.Q\}}$$

Ht-If-False
$$\frac{\{P * v = \mathsf{false}\}\, e_3\, \{u.Q\}}{\{P * v = \mathsf{false}\}\, \mathsf{if}\, v\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3\, \{u.Q\}}$$

Derive Ht-If-True and Ht-If-False from Ht-If. ◊

Exercise 4.10. Show the following derived rule for any expression e and $i \in \{1, 2\}$.

$$\frac{S \vdash \{P\}\, e\, \{v.v = \mathsf{inj}_i\, u * Q\} \qquad S \vdash \{Q[\mathsf{inj}_i\, u/v]\}\, e_i\, [u/x_i]\, \{v.R\}}{S \vdash \{P\}\, \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end}\, \{v.R\}}$$

◊

4.1 Derived rules for Hoare triples, and examples

The following exercises develop some basic building blocks which will be extensively used in proofs later on. They are for the most part simple applications or special cases of the rules in Figure 5 and doing the exercises is good practice to get used to the rules.

Exercise 4.11. Show the following derived rule

$$\textsc{Ht-Pre-Eq}\quad \frac{\Gamma \mid S \vdash \{P[v/x]\}\, e[v/x]\, \{u.Q[v/x]\}}{\Gamma, x : \mathit{Val} \mid S \vdash \{x = v \wedge P\}\, e\, \{u.Q\}}$$

◊

Exercise 4.12. Recall we define the let expression $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ using abstraction and application. Show the following derived rule

$$\textsc{Ht-let}\quad \frac{S \vdash \{P\}\, e_1\, \{x.Q\} \qquad S \vdash \forall v.\, \{Q[v/x]\}\, e_2\, [v/x]\, \{u.R\}}{S \vdash \{P\}\, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2\, \{u.R\}}$$

Using this rule is perhaps a bit inconvenient since most of the time the result of evaluating e_1 will be a single value and the postcondition Q will be of the form $x = v \wedge Q'$ for some value v.

The following rule, a special case of the above rule reflects this common case. Derive the rule from the rule Ht-let.

$$\textsc{Ht-let-det}\quad \frac{S \vdash \{P\}\, e_1\, \{x.x = v \wedge Q\} \qquad S \vdash \{Q[v/x]\}\, e_2\, [v/x]\, \{u.R\}}{S \vdash \{P\}\, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2\, \{u.R\}}$$

Define the sequencing expression $e_1; e_2$ and show the following derived rules

$$\textsc{Ht-seq}\quad \frac{S \vdash \{P\}\, e_1\, \{v.Q\} \qquad S \vdash \{\exists x.\, Q\}\, e_2\, \{u.R\}}{S \vdash \{P\}\, e_1; e_2\, \{u.R\}} \qquad \frac{S \vdash \{P\}\, e_1\, \{\_.Q\} \qquad S \vdash \{Q\}\, e_2\, \{u.R\}}{S \vdash \{P\}\, e_1; e_2\, \{u.R\}}$$

where $\_.Q$ means that Q does not mention the return value. ◊

Exercise 4.13. Recall we defined $\lambda x.e$ to be $\mathsf{rec}\, f(x) = e$ where f is some fresh variable. Show the following derived rule.

$$\textsc{Ht-beta}\quad \frac{S \vdash \{P\}\, e[v/x]\, \{u.Q\}}{S \vdash \{P\}\, (\lambda x.e) v\, \{u.Q\}}$$

◊

Exercise 4.14. Derive the following rule.

$$\textsc{Ht-bind-det}\quad \frac{E \text{ is an eval. context} \qquad S \vdash \{P\}\, e\, \{x.x = u \wedge Q\} \qquad S \vdash \{Q[u/x]\}\, E[u]\, \{w.R\}}{S \vdash \{P\}\, E[e]\, \{w.R\}}$$

◊

When proving examples, one constantly uses the rule of consequence HT-CSQ, the bind rule HT-BIND and its derived versions, such as HT-BIND-DET, and the frame rule HT-FRAME. We now prove a specification of a simple program in detail to show how these structural rules are used. In subsequent examples in the following we will use these rules implicitly most of the time.

Example 4.15. We want to show the following triple

$$\{y \hookrightarrow -\}\,\mathsf{let}\,x = 3\,\mathsf{in}\;\; y \leftarrow x + 2\,\{v.v = () \wedge y \hookrightarrow 5\}.$$

We start by using the rule HT-LET-DET which means we have to show two triples (recalling that equality is a persistent proposition)

$$\{y \hookrightarrow -\}\,3\,\{x.x = 3 * Q\} \qquad\qquad \{Q[3/x]\}\,(y \leftarrow x + 2)\,[3/x]\,\{v.v = () * y \hookrightarrow 5\}$$

for some intermediate proposition Q. We choose Q to be $y \hookrightarrow -$ and using the frame rule HT-FRAME and the value rule HT-RET we have the first triple.

For the second triple we use the deterministic bind rule HT-BIND-DET together with the frame rule. First we prove

$$\{y \hookrightarrow -\}\,3 + 2\,\{v.v = 5 * y \hookrightarrow -\}$$

and then use the rule HT-STORE to prove

$$\{y \hookrightarrow -\}\,y \leftarrow 5\,\{v.v = () * y \hookrightarrow 5\}.$$

■

In the following we will not show all the individual steps of proofs since then proofs become very long and tedious. Instead our basic steps will involve Hoare triples at the level of granularity exemplified by the following exercise.

Exercise 4.16. Show the following triples and entailments in detail.

- $$\{R * \ell \hookrightarrow m\}\,\ell \leftarrow !\ell + 5\,\{v.R * v = () * \ell \hookrightarrow (m + 5)\}$$

- $$\{P\}\,e[m + 5/x]\,\{v.Q\} \vdash \{P * \ell \hookrightarrow m\}\,\mathsf{let}\,x = !\ell + 5\,\mathsf{in}\,e\,\{v.Q * \ell \hookrightarrow m\}$$

- Assuming u does not appear in P and Q show the following entailment.

$$\{P\}\,e[v_1/x]\,\{v.Q\} \vdash \{u = (v_1, v_2) * P\}\,\mathsf{let}\,x = \pi_1\,u\,\mathsf{in}\,e\,\{v.Q\}$$

◊

Example 4.17. A slightly more involved example involving memory locations involves the following function. Let

$$\mathsf{swap} = \lambda x\,y.\mathsf{let}\,z = !\,x\,\mathsf{in}\,x \leftarrow !\,y; y \leftarrow z$$

be the function which swaps the value at two locations.

We would like to specify that this function indeed swaps the values at two locations given. One possible formalisation of this is the following specification.

$$\{\ell_1 \hookrightarrow v_1 * \ell_2 \hookrightarrow v_2\}\,\mathsf{swap}\,\ell_1\,\ell_2\,\{v.v = () \wedge \ell_1 \hookrightarrow v_2 * \ell_2 \hookrightarrow v_1\}.$$

To prove it, we will use HT-LET-DET and hence we need to prove the following two triples:.

$$\{\ell_1 \hookrightarrow v_1 * \ell_2 \hookrightarrow v_2\}\,!\ell_1\,\{v.v = v_1 \wedge \ell_1 \hookrightarrow v_1 * \ell_2 \hookrightarrow v_2\}$$

$$\{\ell_1 \hookrightarrow v_1 * \ell_2 \hookrightarrow v_2\}\,\ell_1 \leftarrow !\ell_2; \ell_2 \leftarrow v_1\,\{v.v = () \wedge \ell_1 \hookrightarrow v_2 * \ell_2 \hookrightarrow v_1\}$$

The first one follows immediately from HT-FRAME and HT-LOAD. For the second we use the sequencing rule HT-SEQ, and hence we need to show the following two triples:

$$\{\ell_1 \hookrightarrow v_1 * \ell_2 \hookrightarrow v_2\}\,\ell_1 \leftarrow !\ell_2\,\{\_.\ell_1 \hookrightarrow v_2 * \ell_2 \hookrightarrow v_2\}$$

$$\{\ell_1 \hookrightarrow v_2 * \ell_2 \hookrightarrow v_2\}\,\ell_2 \leftarrow v_1\,\{v.v = () \wedge \ell_1 \hookrightarrow v_2 * \ell_2 \hookrightarrow v_1\}$$

Recall that $\ell_2 \hookrightarrow v_2$ implies $\ell_2 \hookrightarrow -$. Hence the second specification follows by HT-FRAME, HT-CSQ and HT-STORE.

For the first we need to again use the deterministic bind rule HT-BIND-DET. After that the proof consists of parts analogous to the parts we already explained. ■

Remark 4.18. Note that swap will work even if the locations ℓ_1 and ℓ_2 are the same. Hence another specification of swap is

$$\{\ell_1 \hookrightarrow v_1 \wedge \ell_2 \hookrightarrow v_2\}\,\mathsf{swap}\,\ell_1\,\ell_2\,\{v.v = () \wedge \ell_1 \hookrightarrow v_2 \wedge \ell_2 \hookrightarrow v_1\}.$$

This specification is incomparable (neither of them is derivable from the other) with the previous one with the rules we have. In fact, with the rules we have, we are not able to show this specification. ■

In the following examples we will work with linked lists – chains of reference cells with forward links. In order to specify their behaviour we assume (as explained in Section 3.3) that we have sequences and operations on sequences in our logic together with their expected properties. We write [] for the empty sequence and $x : xs$ for the sequence consisting of an element x and a sequence xs. We define a predicate isList to tie concrete program values to logical sequences. Our representation of linked lists will mimic that of inductively defined lists, which are either empty ($\mathsf{inj}_1()$) or a pointer to a pair of a value and another list ($\mathsf{inj}_2\ell$ where $\ell \hookrightarrow (h,t)$). Notice, however, that this is *not* an inductively defined data structure in the sense known from statically typed funtional languages like ML or Coq – it mimics the shape, but nothing inherently prevents the formation of, *e.g.*, cyclical lists.

The $\mathsf{isList}\,l\,xs$ predicate relates values l to sequences xs; it is defined by induction on xs:

$$\mathsf{isList}\,l[] \equiv l = \mathsf{inj}_1()$$
$$\mathsf{isList}\,l(x : xs) \equiv \exists hd, l'.\, l = \mathsf{inj}_2(hd) * hd \hookrightarrow (x, l') * \mathsf{isList}\,l'\,xs$$

Notice that while our data representation in itself does not ensure the absence of, *e.g.*, cyclic lists, the $\mathsf{isList}\,l\,xs$ predicate above does ensure that the list l is acyclic, because of separation of the hd pointer and the inductive occurrence of the predicate.

Exercise 4.19. Explain why the separating conjunction ensures lists are acyclic. What goes wrong if we used ordinary conjunction? ◊

To specify the next example we assume the map function on sequences in the logic. It is the logical function from sequences to sequences that applies f at every index of the sequence: it is defined by the following two equations

$$\begin{aligned}\mathrm{map}\, f\, [] &\equiv []\\ \mathrm{map}\, f\, (x : xs) &\equiv f\, x : \mathrm{map}\, f\, xs\end{aligned}$$

Example 4.20. We now have all the ingredients to write and specify a simple program on linked lists. Let inc denote the following program that increments all values in a linked list of integers:

$$\begin{aligned}\mathsf{rec}\,\mathsf{inc}(l) = \ & \mathsf{match}\, l\, \mathsf{with}\\ & \mathsf{inj}_1\, x_1 \Rightarrow ()\\ & \mid \mathsf{inj}_2\, x_2 \Rightarrow \mathsf{let}\, h = \pi_1\, !x_2\, \mathsf{in}\\ & \qquad\qquad \mathsf{let}\, t = \pi_2\, !x_2\, \mathsf{in}\\ & \qquad\qquad x_2 \leftarrow (h+1, t);\\ & \qquad\qquad \mathsf{inc}\ t\\ & \mathsf{end}\end{aligned}$$

We wish to give it the following specification:

$$\forall xs.\forall l.\{\mathrm{isList}\, l xs\}\, \mathsf{inc}\, l\, \{v.v = () \wedge \mathrm{isList}\, l(\mathrm{map}(1+)xs)\}$$

We proceed by the Ht-Rec rule and we consider two cases: we use the fact that the sequence xs is either empty [] or has a head x followed by another sequence xs'.

In the first case we need to show

$$\forall l.\{\mathrm{isList}\, l[]\}\, \mathsf{match}\ l\ \mathsf{with}\ \ldots\, \{v.v = () \wedge \mathrm{isList}\, l(\mathrm{map}(1+)[])\}$$

and in the second

$$\forall x, xs'.\forall l.\{\mathrm{isList}\, l(x : xs')\}\, \mathsf{match}\, l\, \mathsf{with}\ldots\, \{v.v = () \wedge \mathrm{isList}\, l(\mathrm{map}(1+)(x : xs'))\}$$

where in the body we have replaced inc with the function f for which we assume the triple (the assumption of the premise of the Ht-Rec rule)

$$\forall xs.\forall l.\{\mathrm{isList}\, l xs\}\, f\, l\, \{v.v = () \wedge \mathrm{isList}\, l(\mathrm{map}(1+)xs)\}. \tag{3}$$

In both cases we proceed by the derived match rule from Exercise 4.10, as the isList predicate tells us enough information to determine the chosen branch of the match statement.

In the first case we have $\mathrm{isList}\, l[] \equiv l = \mathsf{inj}_1\, ()$, and thus we easily prove

$$\{\mathrm{isList}\, l[]\}\, l\, \{v.v = \mathsf{inj}_1\, () * \mathrm{isList}\, l[]\}$$

by Ht-frame and Ht-Pre-Eq followed by Ht-ret.

Thus we know the first branch is taken and so according to the derived match rule from Exercise 4.10 we need to show the following triple.

$$\{\mathrm{isList}\, l[]\}\, ()\, \{v.v = () * \mathrm{isList}\, l(\mathrm{map}(+1)[])\}$$

which follows by Ht-ret after framing away $\mathrm{isList}\, l[]$, which is allowed as $\mathrm{map}(+1)[] \equiv []$.

In the case where the sequence is not empty we have that

$$\mathrm{isList}\, l (x : xs) \equiv \exists hd, l'. l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l') * \mathrm{isList}\, l' xs'$$

and thus we can prove

$$\begin{array}{l} \{l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l') * \mathrm{isList}\, l' xs'\} \\ \quad l \\ \{r.r = \mathsf{inj}_2\, hd * l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l') * \mathrm{isList}\, l' xs'\} \end{array}$$

for some l' and hd, using the rule Ht-exist, the frame rule, the Ht-Pre-Eq rule, and the Ht-ret rule.

Exercise 4.21. Prove this Hoare triple in detail using the rules just mentioned. ◊

This is enough to determine that the match takes the second branch, and using the derived match rule from Exercise 4.10 we proceed to verify the body of the second branch. Using the rules Ht-let-det and Ht-Proj repeatedly we quickly prove

$$\begin{array}{l} \{l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l') * \mathrm{isList}\, l' xs'\} \\ \quad \mathsf{let}\, h = \pi_1\, !hd\, \mathsf{in} \\ \quad \mathsf{let}\, t = \pi_2\, !hd\, \mathsf{in} \\ \quad hd \leftarrow (h + 1, t) \\ \{l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x + 1, l') * \mathrm{isList}\, l' xs' * t = l' * h = x\} \end{array}$$

Now

$$l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x + 1, l') * \mathrm{isList}\, l' xs' * t = l' * h = x$$

clearly implies

$$l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x + 1, l') * \mathrm{isList}\, t xs'$$

and thus, by the sequencing rule Ht-seq and the rule of consequence Ht-csq, we are left with proving

$$\begin{array}{l} \{l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x + 1, l') * \mathrm{isList}\, t xs'\} \\ \quad f\, t \\ \{r.r = () * \mathrm{isList}\, l (\mathrm{map}(+1)(x : xs'))\} \end{array}$$

which follows from the induction hypothesis (3), and the definition of the isList predicate. ■

Remark 4.22 (About functions taking multiple arguments)**.** The programming language $\lambda_{\mathrm{ref,conc}}$ only has as primitive functions which take a single argument. Functions taking multiple arguments can be encoded as either higher-order functions returning functions, or as functions taking tuples as arguments.

Therefore we use some syntactic sugar to write functions taking multiple arguments in a more readable way. We write

$$\mathsf{rec}\, f (x, y) = e$$

for the following term

$$\mathsf{rec}\, f (p) = \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, \mathsf{in}\, e.$$

This notation is generalised in an analogous way to three and more arguments. The corresponding derived Hoare rule is the following

$$\textsc{Ht-Rec-multi}\quad \frac{\Gamma, g : \mathit{Val} \mid S \wedge \forall z.\, \forall v_1.\, \forall v_2.\, \{P\}\, g(v_1, v_2)\, \{u.Q\} \vdash \forall z.\, \forall v_1.\, \forall v_2.\, \{P\}\, e[g/f, v_1/x, v_2/y]\, \{u.Q\}}{\Gamma \mid S \vdash \forall z.\, \forall v.\, \{\exists v_1, v_2.\, v = (v_1, v_2) \wedge P\}\, (\mathsf{rec}\, f(x,y) = e) v\, \{u.\exists v_1, v_2.\, v = (v_1, v_2) \wedge Q\}}$$

where we assume that the variable v is fresh for P and Q. ■

Exercise 4.23. Derive the rule Ht-Rec-multi. ◊

Exercise 4.24. Let append be the following function, which takes two linked lists as arguments and returns a list which is the concatenation of the two.

$$\begin{array}{l}
\mathsf{rec}\ \mathsf{append}(l, l') = \mathsf{match}\, l\, \mathsf{with} \\
\quad \mathsf{inj}_1\, x_1 \Rightarrow l' \\
\quad \mid \mathsf{inj}_2\, x_2 \Rightarrow \mathsf{let}\, p = !\, x_2\, \mathsf{in} \\
\qquad\qquad \mathsf{let}\, r = \mathsf{append}\, (\pi_2\, p)\, l'\, \mathsf{in} \\
\qquad\qquad x_2 \leftarrow (\pi_1\, p, r); \\
\qquad\qquad \mathsf{inj}_2\, x_2 \\
\quad \mathsf{end}
\end{array}$$

We wish to give it the following specification where $+\!\!+$ is append on mathematical sequences.

$$\forall xs, ys, l, l'.\{\mathsf{isList}\, l\, xs * \mathsf{isList}\, l'\, ys\}\, \mathsf{append}\, l\, l'\, \{v.\, \mathsf{isList}\, v\, (xs +\!\!+ ys)\}.$$

- Prove the specification.
- Is the following specification also valid?

  $$\forall xs, ys, l, l'.\{\mathsf{isList}\, l\, xs \wedge \mathsf{isList}\, l'\, ys\}\, \mathsf{append}\, l\, l'\, \{v.\, \mathsf{isList}\, v\, (xs +\!\!+ ys)\}$$

  Hint: Think about what is the result of $\mathsf{append}\, l\, l$.

◊

Exercise 4.25. The append function in the previous exercise is not tail recursive and hence it space consumption is linear in the length of the first list. A better implementation of append for linked lists is the following.

$$\begin{array}{l}
\mathsf{append'}\ l\ l' = \mathsf{let}\, go = \mathsf{rec}\, f(h\, p) = \mathsf{match}\, h\, \mathsf{with} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{inj}_1\, x_1 \Rightarrow p \leftarrow (\pi_1\, (!\, p), l') \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{inj}_2\, x_2 \Rightarrow f\, (\pi_2\, (!\, x_2))\, x_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{end} \\
\quad \mathsf{in}\ \mathsf{match}\, l\, \mathsf{with} \\
\qquad \mathsf{inj}_1\, x_1 \Rightarrow l' \\
\qquad \mid \mathsf{inj}_2\, x_2 \Rightarrow go\ l\ x_2; \\
\qquad\qquad\qquad\quad l \\
\qquad \mathsf{end}
\end{array}$$

In the function go the value p is the last node of the list l we have seen while traversing l. Thus go traverses the first list and once it reaches the end it updates the tail pointer in the last node to point to the second list, l'.

Prove for append′ the same specification as for append above. You need to come up with a strong enough invariant for the function go, relating h, p and xs and ys. ◊

Exercise 4.26. Implement, specify and prove a length function for linked lists. ◊

Exercise 4.27. Using the above specifications, construct a program using append and length, give it a reasonable specification and prove it. ◊

For the following example, we need to relate a linked list to the reversal of a mathematical sequence. The reverse function on sequences is defined as follows:

$$\begin{aligned}\mathrm{reverse}[] &\equiv [] \\ \mathrm{reverse}(x : xs) &\equiv \mathrm{reverse}\, xs +\!\!+ [x]\end{aligned}$$

Example 4.28. Consider the following program which performs the in-place reversal of a linked list using an accumulator to remember the last element which the function visited:

$$\begin{aligned}\mathsf{rec}\,\mathsf{rev}(l, acc) = \ & \mathsf{match}\, l\, \mathsf{with} \\ & \mathsf{inj}_1\, x_1 \Rightarrow acc \\ & \mid \mathsf{inj}_2\, x_2 \Rightarrow \mathsf{let}\, h = \pi_1\, !\, x_2\, \mathsf{in} \\ & \qquad\qquad \mathsf{let}\, t = \pi_2\, !\, x_2\, \mathsf{in} \\ & \qquad\qquad x_2 \leftarrow (h, acc); \\ & \qquad\qquad \mathsf{rev}(t, l) \\ & \mathsf{end}\end{aligned}$$

Intuitively we wish to give it the following specification:

$$\forall vs. \forall hd. \{\mathrm{isList}\, hd\ vs\}\, \mathsf{rev}(hd, \mathsf{inj}_1\, ())\, \{r. \mathrm{isList}\, r\ (\mathrm{reverse}\, vs)\}$$

The resulting induction hypothesis is, however, not strong enough, and we need to *strengthen* the specification. On the one hand, we are now proving a stronger statement, which requires more work. On the other, we get to leverage a much stronger induction hypothesis, eventually allowing the proof to go through.

We generalize the specification of rev to the following specification:

$$\forall vs, us. \forall hd, acc. \{\mathrm{isList}\, hd\ vs * \mathrm{isList}\, acc\ us\}\, \mathsf{rev}(hd, acc)\, \{r. \mathrm{isList}\, r (\mathrm{reverse}\, vs +\!\!+ us)\}$$

Exercise 4.29. Consider a tail recursive implementation of reverse on mathematical sequences, as defined by the following equations:

$$\begin{aligned}\mathrm{reverse'}[] acc &\equiv acc \\ \mathrm{reverse'}(x : xs) acc &\equiv \mathrm{reverse'}\, xs (x : acc)\end{aligned}$$

Convince yourself that $\forall xs. \mathrm{reverse'}\, xs[] \equiv \mathrm{reverse}\, xs$ is not directly provable by induction as the resulting induction hypothesis is too weak. ◊

Exercise 4.30. Prove that in-place reversing twice yields the original list. ◊

We here proceed with the proof of the general specification. The strategy is the same as in the case of the inc function: we use the Ht-Rec rule and case analysis on the sequence vs, followed by the derived match rule.

If $vs = []$, we argue that

$$\{\text{isList}\, l\ []\ast\text{isList}\, acc\ us\}\, l\, \{r.r = \mathsf{inj}_1\,()\ast\text{isList}\, acc\ us\}$$

and thus by using the derived rule for match from Exercise 4.10 we need to show the following triple for the first branch of the match.

$$\{\text{isList}\, acc\ us\}\, acc\, \{r.\,\text{isList}\, r(\text{reverse}[]+\!\!+ us)\}$$

This is easily seen to hold as $\text{reverse}[]+\!\!+ us \equiv us$.

If $vs = v : vs'$ for some v and vs', then we see, by unfolding the isList predicate, that there exists l'', hd such that

$$\{\text{isList}\, l\ (v : vs')\ast\text{isList}\, acc\ us\}$$

$$l$$

$$\{r.r = \mathsf{inj}_2\, hd \ast l = r \ast hd \hookrightarrow (v, l')\ast\text{isList}\, l'\ vs'\ast\text{isList}\, acc\ us\}$$

and we are thus in the second branch of the match. We start by showing

$$\begin{aligned}&\{l = \mathsf{inj}_2\, hd \ast hd \hookrightarrow (v, l')\ast\text{isList}\, l'\ vs'\ast\text{isList}\, acc\ us\}\\ &\quad\mathsf{let}\, h = \pi_1\,!hd\ \mathsf{in}\\ &\quad\mathsf{let}\, t = \pi_2\,!hd\ \mathsf{in}\\ &\quad hd \leftarrow (h, acc)\\ &\{l = \mathsf{inj}_2\, hd \ast hd \hookrightarrow (v, acc)\ast\text{isList}\, l'\ vs'\ast\text{isList}\, acc\ us\ast h = v \ast t = l'\}\end{aligned}$$

by repeated applications of Ht-let-det and Ht-Proj. Clearly the proposition

$$l = \mathsf{inj}_2\, hd \ast hd \hookrightarrow (v, acc)\ast\text{isList}\, l'\ vs'\ast\text{isList}\, acc\ us\ast h = v \ast t = l'$$

implies the proposition

$$l = \mathsf{inj}_2\, hd \ast hd \hookrightarrow (v, acc)\ast\text{isList}\, t\ vs'\ast\text{isList}\, acc\ us\ast h = v$$

which is simply

$$\text{isList}\, t\ vs'\ast\text{isList}\, l\ (v : us)$$

by definition of the isList predicate. Finally by using the induction hypothesis (the assumption of Ht-Rec) we have

$$\begin{aligned}&\{\text{isList}\, t\ vs'\ast\text{isList}\, l\ (v : us)\}\\ &\quad\mathsf{rev}(t, l)\\ &\{r.\,\text{isList}\, r(\text{reverse}\, vs'+\!\!+ v : vs)\}\end{aligned}$$

and we are done, observing that $(\text{reverse}\, vs'+\!\!+ v : vs) \equiv \text{reverse}(v : vs')+\!\!+ vs$.

Thus combining all of these using the sequencing rule HT-SEQ and the rule of consequence HT-CSQ we have proved

$$\begin{array}{l}\{l = \mathsf{inj}_2\, hd * hd \hookrightarrow (v, l') * \mathrm{isList}\, l'\ vs' * \mathrm{isList}\, acc\ us\} \\ \quad \mathsf{let}\, h = \pi_1\, !hd\, \mathsf{in} \\ \quad \mathsf{let}\, t = \pi_2\, !hd\, \mathsf{in} \\ \quad hd \leftarrow (h, acc); \mathsf{rev}(t, l) \\ \{r.\, \mathrm{isList}\, r(\mathrm{reverse}(v : vs') +\!\!+ vs)\}\end{array}$$

as required. ■

4.2 Abstract data types

Suppose we wish to write and specify a simple counter module. There are several ways of writing it and specifying it.

The simplest way is to write the following three functions

$$\begin{aligned}\mathrm{mk\_counter} &= \lambda\_.\mathsf{ref}(0) \\ \mathrm{inc\_counter} &= \lambda x.x \leftarrow\, !\, x + 1 \\ \mathrm{read\_counter} &= !\, x\end{aligned}$$

and prove the following three specifications

$$\begin{array}{l}\{\mathsf{True}\}\ \mathrm{mk\_counter}()\,\{v.v \hookrightarrow 0\} \\ \{\ell \hookrightarrow n\}\ \mathrm{inc\_counter}\,\ell\,\{v.v = () \wedge \ell \hookrightarrow n+1\} \\ \{\ell \hookrightarrow n\}\ \mathrm{read\_counter}\,\ell\,\{v.v = n \wedge \ell \hookrightarrow n\}.\end{array}$$

Exercise 4.31. Prove the above three specifications. ◊

The above specification is unsatisfactory since it completely exposes the internals of the counter, which means that it is not modular: if we verify a client of the counter relative to the above specification and then change the implementation of the counter, then the specification will likely also change and we will have to re-verify the client. A more abstract and modular specification is the following, which existentially quantifies over the "counter representation predicate" C, thus hiding the fact that the return value is a location.

$$\begin{array}{l}\exists C : \mathit{Val} \to \mathbb{N} \to \mathsf{Prop}. \\ \{\mathsf{True}\}\ \mathrm{mk\_counter}()\,\{c.C(c,0)\} * \\ \forall c.\, \{C(c,n)\}\ \mathrm{inc\_counter}\, c\,\{v.v = () \wedge C(c,n+1)\} * \\ \forall c.\, \{C(c,n)\}\ \mathrm{read\_counter}\, c\,\{v.v = n \wedge C(c,n)\}.\end{array} \tag{4}$$

This approach is not ideal either, because the code, consisting of three separate functions, does not provide any abstraction on its own (a client of this code would be able to modify directly the contents of the reference cell reprensenting the counter rather than only through the read and increment methods) and is not the kind of code that realistically would be written. In a typed language, the three functions would typically be sealed in a module, and the return type of the mk_counter method would be abstract, only supporting read and write operations.

In our untyped language, we can also hide the internal state and only expose the read and increment methods as follows:

$$\text{counter} = \lambda\_.\mathsf{let}\, x = \mathsf{ref}(0)\,\mathsf{in}\,(\lambda\_.x \leftarrow !\,x + 1,\ \lambda\_.!\,x)$$

The idea is that the counter method returns a pair of methods which have a hidden internal state variable x. The specification of this counter needs nested Hoare triples. One possibility is the following. To aid readability we use $v.\mathrm{inc}$ in place of $\pi_1\, v$ and analogously for $v.\mathrm{read}$.

$$\{\mathsf{True}\}\ \mathrm{counter}()\left\{v.\exists \ell.\ \begin{array}{l} \ell \hookrightarrow 0 * \\ \forall n.\,\{\ell \hookrightarrow n\}\,v.\,\mathrm{inc}()\,\{u.u = () \land \ell \hookrightarrow (n+1)\} * \\ \forall n.\,\{\ell \hookrightarrow n\}\,v.\,\mathrm{read}()\,\{u.u = n \land \ell \hookrightarrow n\} \end{array}\right\}$$

A disadvantage of this specification is, again, that it exposes the fact that the internal state is a single location which contains the value of the counter.

Exercise 4.32. Show this specification of the counter method. ◊

A better, modular, specification completely hides the internal state in an abstract predicate:

$$\{\mathsf{True}\}\ \mathrm{counter}()\left\{v.\exists C : \mathbb{N} \to \mathsf{Prop}.\ \begin{array}{l} C(0) * \\ \forall n.\,\{C(n)\}\,v.\,\mathrm{inc}()\,\{u.u = () \land C(n+1)\} * \\ \forall n.\,\{C(n)\}\,v.\,\mathrm{read}()\,\{u.u = n \land C(n)\} \end{array}\right\} \tag{5}$$

Exercise 4.33. Show this specification of the counter method. Can you derive it from the previous one? What about conversely? Can you derive the previous specification from the current one? Hint: Think about another implementation of counter which satisfies the second specification, but not the first. ◊

Exercise 4.34. Define the counter method with the help of the methods mk_counter, inc_counter and read_counter and derive specification (5) from specification (4). ◊

Ideally we would want to use this combination of code and specification. However, it has some practical usability downsides when used in the accompanying Coq formalization. Chief among them is that it is easier to define, *e.g.*, an **is_counter** predicate, and use that instead of always having to deal with eliminating an existentially quantified predicate. To make the proofs more manageable, we will therefore usually write modules and specifications in the style of (4) and understand that we let go of abstraction at the level of the programming language.

As long as we are only using the specifications of programs it does not matter that the actual implementation exposes more details. And as we will see in examples, this is how we prove clients of modules. When working in Coq, for instance, we can ensure this mode of use of code and specifications using Coq's abstraction features.

4.3 Abstract data types and ownership transfer: a stack module

We wish to specify a stack module with three methods, mk_stack for creating the stack, and push and pop methods for adding and removing elements from the top of the stack. Drawing

inspiration from the previous section, we might come up with the following specification of the stack module.

$$\begin{aligned}&\exists \mathrm{isStack} : \mathit{Val} \to \mathrm{list}\,\mathit{Val} \to \mathsf{Prop}.\\ &\quad \{\mathsf{True}\}\ \mathrm{mk\_stack}()\,\{s.\mathrm{isStack}(s,[])\} \land\\ &\quad \forall s.\forall xs.\{\mathrm{isStack}(s,xs)\}\ \mathrm{push}(x,s)\,\{v.v = \mathsf{inj}_1\,() \land \mathrm{isStack}(s,x:xs)\} \land\\ &\quad \forall s.\forall x,xs.\{\mathrm{isStack}(s,x:xs)\}\ \mathrm{pop}(s)\,\{v.v = \mathsf{inj}_2\,x \land \mathrm{isStack}(s,xs)\}\end{aligned} \tag{6}$$

Here $\mathrm{isStack}(s,xs)$ is an assertion stating that s is a stack, which contains the list of elements xs (in the specified order). The specifications of push and pop are self-explanatory. This specification is useful for verification of certain clients, namely those that push and pop pure values, such as numbers, strings, and integers, to and from the stack. However, it is not strong enough to verify clients which operate on stacks of, *e.g.*, locations.

Exercise 4.35. Using the stack specification above, show that the program e defined as

$$\mathsf{let}\, s = \mathrm{mk\_stack}()\,\mathsf{in}\ \mathrm{push}(1,s);\mathrm{push}(2,s);\mathrm{pop}(s);\mathrm{pop}(s)$$

satisfies the specification

$$\{\mathsf{True}\}\, e\, \{v.v = \mathsf{inj}_2\, 1\}.$$

◊

The reason the stack specification above suffices for reasoning about stack clients that push and pop pure values is that all relevant information about a pure value is contained in the value itself. In contrast, the meaning of other values, such as locations, depends on other features, *e.g.*, for the case of locations, the heap. Thus to be able to reason about clients that push and pop other values than pure values, we need a specification which allows us to transfer this additional information to and from the stack. To this end, we change the specification such that the isStack predicate does not specify a list of values directly but rather their properties, in the form of Iris predicates which hold for the values stored in the stack.

The new specification of the stack module is thus as follows.

$$\begin{aligned}&\exists \mathrm{isStack} : \mathit{Val} \to \mathrm{list}(\mathit{Val} \to \mathsf{Prop}) \to \mathsf{Prop}.\\ &\quad \{\mathsf{True}\}\ \mathrm{mk\_stack}()\,\{s.\mathrm{isStack}(s,[])\} \land\\ &\quad \forall s.\forall \Phi.\forall \Phi s.\{\mathrm{isStack}(s,\Phi s) * \Phi(x)\}\ \mathrm{push}(x,s)\,\{v.v = () \land \mathrm{isStack}(s,\Phi : \Phi s)\} \land\\ &\quad \forall s.\forall \Phi.\forall \Phi s.\{\mathrm{isStack}(s,\Phi : \Phi s)\}\ \mathrm{pop}(s)\,\{v.\Phi(v) * \mathrm{isStack}(s,\Phi s)\}\end{aligned} \tag{7}$$

There are several things to notice about this specification. First is the universal quantification over arbitrary predicates Φ and lists of predicates Φs, this makes the specification *higher-order*.

The second thing to notice is the *ownership transfer* in the specifications of push and pop methods. This is related to the ownership reading of assertions explained in Section 3.1 above. The idea is that, when pushing, the client of the stack transfers the resources associated with the element x (the assertion $\Phi(x)$) to the stack. Conversely, when executing the pop operation the client of the stack gets the value v and the resources associated with the value (the assertion $\Phi(v)$).

An example of a resource that can be transferred is the points to assertion $\ell \hookrightarrow 3$. In this case, the Φ predicate would be instantiated with $\lambda x.x \hookrightarrow 3$.

Exercise 4.36. Derive specification (6) from the more general specification (7). ◊

To see where this more general specification is useful let us consider an example client.

$$
\begin{array}{l}
\mathsf{let}\, s = \mathrm{mk\_stack}()\, \mathsf{in} \\
\mathsf{let}\, x_1 = \mathsf{ref}(1)\, \mathsf{in} \\
\mathsf{let}\, x_2 = \mathsf{ref}(2)\, \mathsf{in} \\
\mathrm{push}(x_1, s); \mathrm{push}(x_2, s); \mathrm{pop}(s); \\
\mathsf{let}\, y = \mathrm{pop}(s)\, \mathsf{in} \\
\mathsf{match}\, y\, \mathsf{with} \\
\quad \mathsf{inj}_1\, () \Rightarrow () \\
\mid \mathsf{inj}_2\, \ell \Rightarrow\, !\ell \\
\mathsf{end}
\end{array}
\tag{8}
$$

Exercise 4.37. Show the following specification of the program (8).

$$\{\mathsf{True}\}\,(8)\,\{v.v = 1\}.$$

◊

The stronger specification also gives us additional flexibility when specifying functions working with stacks. For instance, we can specify a function f which expects a stack of locations pointing to prime numbers and returns a prime number as follows.

$$\{\mathrm{isStack}(s, \Phi s) * \mathrm{PrimeLoc}(\Phi s)\}\, f\, s\, \{v.\, \mathrm{isPrime}(v)\}$$

where $\mathrm{PrimeLoc}(\Phi)$ asserts that all predicates in Φs are of a particular form, namely that they hold only for locations which point to prime numbers. And $\mathrm{isPrime}(v)$ asserts that v is a prime number. We omit its definition here.

The assertion $\mathrm{PrimeLoc}(\Phi s)$ can be defined by recursion on the list Φs by the following two cases.

$$
\begin{aligned}
\mathrm{PrimeLoc}([]) &\equiv \mathsf{True} \\
\mathrm{PrimeLoc}(\Phi : \Phi s) &\equiv (\Phi(v) \mathrel{-\!\!*} \exists n.\, v \mapsto n * \mathrm{isPrime}(n)) * \mathrm{PrimeLoc}(\Phi s)
\end{aligned}
$$

Such a use case where we want a certain property to hold for all elements of a stack is common. Thus it is useful to have a less general stack module specification tailored for the use case. The specification we have in mind is

$$
\begin{array}{l}
\exists \mathrm{isStack} : \mathit{Val} \to \mathrm{list}\mathit{Val} \to (\mathit{Val} \to \mathsf{Prop}) \to \mathsf{Prop}. \\
\forall \Phi : \mathit{Val} \to \mathsf{Prop}. \\
\quad \{\mathsf{True}\}\ \mathrm{mk\_stack}()\, \{s.\mathrm{isStack}(s, [], \Phi)\} \wedge \\
\quad \forall s. \forall xs. \{\mathrm{isStack}(s, xs, \Phi) * \Phi(x)\}\ \mathrm{push}(x, s)\, \{v.v = () \wedge \mathrm{isStack}(s, x : xs, \Phi)\} \wedge \\
\quad \forall s. \forall x, xs. \{\mathrm{isStack}(s, x : xs, \Phi)\}\ \mathrm{pop}(s)\, \{v.v = x \wedge \mathrm{isStack}(s, xs, \Phi) * \Phi(x)\}
\end{array}
\tag{9}
$$

The idea is that $\mathrm{isStack}(s, xs, \Phi)$ asserts that s is a stack whose values are xs and all of the values $x \in xs$ satisfy the given predicate Φ. The difference from the specification (7) is that there is a uniform predicate Φ which *all* the elements have to satisfy, as opposed to having a list of predicates, one for each element of the stack. We can derive the specification (9) from the specification (7) as follows.

Let $\text{isStack}_g : \mathit{Val} \to \text{list}(\mathit{Val} \to \text{Prop}) \to \text{Prop}$ be the predicate associated with the general stack specificaton (7). We define $\text{isStack}_u : \mathit{Val} \to \text{list}\mathit{Val} \to (\mathit{Val} \to \text{Prop}) \to \text{Prop}$ as

$$\text{isStack}_u(s, xs, \Phi) \equiv \text{isStack}_g(s, \Phi s(xs))$$

where the list Φs is defined recursively from the list xs as

$$\begin{aligned} \Phi s([]) &\equiv [] \\ \Phi s(x : xs) &\equiv (\lambda y.x = y \wedge \Phi(y)) : \Phi s(xs) \end{aligned}$$

Exercise 4.38. Using isStack_u derive the specifications of mk_stack, push and pop as stated in (9). ◊

4.4 Case Study: foldr

In this section we will use the logic introduced thus far to give a very general higher order specification of the foldr function on linked lists. We then use this specification to verify two clients using the foldr function, the sumList function, which computes the sum of all the elements in the linked list, and the filter function, which creates a new list with only those elements of the original list which satisfy the given predicate.

The implementation of the foldr function is as follows.

$$\begin{aligned} \text{rec}\,\text{foldr}(f, a, l) = \ & \text{match}\, l\, \text{with} \\ & \quad \text{inj}_1\, x_1 \Rightarrow a \\ & \mid \text{inj}_2\, x_2 \Rightarrow \text{let}\, h = \pi_1\, !\, x_2\, \text{in} \\ & \qquad\qquad \text{let}\, t = \pi_2\, !\, x_2\, \text{in} \\ & \qquad\qquad f\,(h, \text{foldr}(f, a, t)) \\ & \text{end} \end{aligned}$$

The first argument f is a function taking a pair as an argument, the second argument a is the result of foldr on the empty list, and l is the linked list.

The specification of the foldr function is as follows.

$$\begin{aligned} \forall P, I.\, \forall f \in \mathit{Val}.\, \forall xs.\, \forall l.\, \{&\text{isList}\, l\, xs * \text{all}\, P\, xs * I\, [] * a * (\forall x \in \mathit{Val}.\, \forall a' \in \mathit{Val}.\, \forall ys.\, \{P x * I\, ys\, a'\}\, f\, (x, a')\, \{r. I(x : ys) r\})\} \\ & \text{foldr}(f, a, l) \\ & \{r.\, \text{isList}\, l\ xs * I\, xs\, r\} \end{aligned}$$

where the predicate $\text{all}\, P\, xs$ states that P holds for all elements of the list. It is defined by induction on the list as

$$\begin{aligned} \text{all}\, P\, [] &\equiv \text{True} \\ \text{all}\, P\, (x : xs) &\equiv P\, x * \text{all}\, P\, xs \end{aligned}$$

We want the specification of foldr to capture to following:

- The third argument l needs to be linked list implementing the mathematical sequence xs. This it captured by $\text{isList}\ l\ xs$ in the precondition.

- We do not want foldr to change the list. This is captured by having isList l xs in the postcondition.
- Some clients may not want to operate on general lists but only on subsets of those (*e.g.*, only on lists of odd natural numbers or lists of booleans). This is captured by letting the client choose a predicate P and then having all $P\,xs$ in the precondition force the specification to only hold for list xs, where $P\ x$ holds for all x in xs.
- We want to be able to relate the return value r to the list xs we have folded. This is done by an invariant $I\,xs\,r$. In the case were foldr is applied to the empty list the return value is simply the a provided, hence I needs to be chosen such that $I\,[]\,a$ holds. This is expressed by the assertion $I\,[]\,a$ in the precondition.

  The last assertion in the precondition is the specification of the function f which is used to combine the elements of the list, we remark more on its specification below.

The specification of foldr is higher-order in the sense that it involves nested Hoare triples (here in the precondition). The reason being that foldr takes a function f as argument, hence we can't specify foldr without having some knowledge or specification for the function f. Different clients may instantiate foldr with some very different functions, hence it can be hard to give a specification for f that is reasonable and general enough to support all these choices. In particular knowing when one have found a good and provable specification can be difficult in itself.

The specification for f that we have chosen is

$$\forall x.\,\forall a'.\,\forall ys.\,\{P x * I\ ys\ a'\}\,f\ (x,\ a')\,\{r.I\ (x :: ys)\ r\}$$

We explain the specification by how it is used proof of foldr, which is detailed below. In the proof of foldr we are only going to use the specification for f when l represents the sequence $(x :: xs)$, where the sequence xs is represented by some implementation t. In this case we are going to instantiate the specification of f with foldr(f, a, t) as a', xs as ys and x as x.

In this sense the specification for f simply says that if x satisfies P and the result a' of folding f over the subsequence xs satisfies the invariant, then applying f to (x, a') (which is exactly what foldr does on the sequence $(x :: xs)$) gives a result, such that I relates it to the sequence $(x :: xs)$.

Remark 4.39. The only place, where our concrete implementation l of the list (as a linked list) is used is in isList $l\,xs$ where it is linked to a mathematical sequence xs. The predicates P, I and all are all defined on mathematical sequences instead. In this sense we may say that the mathematical sequences are abstractions or models of our concrete lists and P, I and all operate on these models, hence the distinction between implementation details and mathematical properties becomes clear. ■

Proof of the specification

Let P, I, f, xs and l be given. As Hoare triples are persistent and persistence is preserved by quantification we may move

$$\forall x.\,\forall a'.\,\forall ys.\,\{P x * I\,ys\,a'\}\,f\,(x, a')\,\{r.I\,(x : ys)\}$$

into the context i.e. we may assume it. We thus have to prove

$$\forall x.\,\forall a'.\,\forall ys.\,\{P\,x * I\,ys\,a'\}\,f\,(x, a')\,\{r.I\,(x : ys)\} \vdash \begin{array}{l}\{\text{isList}\ l\,xs * \text{all}\,P\,xs * I\,[]\,a\} \\ \quad \text{foldr}(f, a, l) \\ \{r.\,\text{isList}\ l\,xs * I\,xs\,r\}\end{array}$$

We proceed by Ht-Rec i.e. we have to prove the specification for the body of foldr in which we have replaced any occurrence of foldr with the function g for which

$$\forall xs.\, \forall l.\, \{\text{isList}\; l\; xs * \text{all}\; P\; xs * I\; []\; a\}\; g(f, a, l)\; \{r.\, \text{isList}\; l\; xs * I\; xs\; r\}$$

is assumed.

By the definition of the isList predicate we have that the list l points to is either empty [] or has a head x followed by another list xs'.

In the first case we need to show

$$\{\text{isList}\; l\; [] * \text{all}\; P\; [] * I\; []\; a\}\; \text{match}\, l\, \text{with} \ldots \{r.\, \text{isList}\; l\; [] * I\; []\; r\}$$

and in the second

$$\{\text{isList}\; l\; (x : xs') * \text{all}\; P\; (x : xs') * I\; []\; a\}\; \text{match}\, l\, \text{with} \ldots \{r.\, \text{isList}\; l\; xs * I\; xs\; r\}$$

In both cases we proceed by the derived match rule from Exercise 4.10.

In the first case we have $\text{isList}\; l\; [] \equiv l = \text{inj}_1()$ hence the first case is taken (follows by Ht-frame, Ht-Pre-Eq and Ht-ret), thus we have to prove

$$\{\text{isList}\; l\; [] * I\; []\; a\}\; a\; \{r.\, \text{isList}\; l\; [] * I\; []\; r\}$$

After framing $\text{isList}\; l\; []$ we are left with proving

$$\{I\; []\; a\}\; a\; \{r.\, I\; []\; r\}$$

As $I\; []\; r$ follows from $r = a * I\; []\; a$ it suffices by Ht-csq to show

$$\{I\; []\; a\}\; a\; \{r.\, r = a * I\; []\; a\}$$

which follows by Ht-frame and Ht-ret.

In the second case we have $\text{isList}\; l\; (x : xs') \equiv \exists hd, l'.\, l = \text{inj}_2 hd * hd \hookrightarrow (x, l') * \text{isList}\; l'\; xs'$. By exercise 4.21 we get that the second case is taken, thus we need to prove

$$\begin{array}{l}\{\text{isList}\; l'\; xs' * \text{all}\; P\; (x : xs') * I\; []\; a * l = \text{inj}_2\; hd * hd \hookrightarrow (x, l')\} \\ \quad \text{let}\, h = \pi_1\; !\, x_2\; \text{in} \\ \quad \text{let}\, t = \pi_2\; !\, x_2\; \text{in} \\ \quad f\; (h, (g(f, a, t))) \\ \{r.\, \text{isList}\; l\; (x : xs') * I\; (x : xs')\; r\}\end{array}$$

By simple applications of Ht-let-det, Ht-frame, Ht-bind, Ht-load, Ht-Proj and Ht-ret we need to show:

$$\begin{array}{l}\{\text{isList}\; l'\; xs' * \text{all}\; P\; (x : xs) * I\; []\; a * l = \text{inj}_2\; hd * hd \hookrightarrow (x, l')\} \\ \quad f(x, g(f, a, l')) \\ \{r.\, \text{isList}\; l\; (x : xs')\; r\}\end{array}$$

By Ht-bind-det we have to show:

1. $f\; (x, -)$ is an evaluation context.

2. the specification

$$\{\text{isList } l'\, xs' * \text{all } P\, (x : xs') * I\, [\,]\, a * l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l')\}$$
$$g(f, a, l')$$
$$\{r.\, \text{isList } l'\, xs' * I\, xs'\, r * P\, x * l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l')\}$$

3. the specification

$$\{\text{isList } l'\, xs' * I\, xs'\, v * P\, x * l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l')\}$$
$$f(x, v)$$
$$\{r.\, \text{isList } l\, (x : xs') * I\, (x : xs')\, r\}$$

1. As f is a value by assumption it follows that $f\, E$ is an evaluation context, when E is. Now as x is a value we have that $(x, -)$ is an evaluation context hence it follows that $f(x, -)$ is an evaluation context.[3]
2. Follows by unfolding the definition of $\text{all } P(x : xs') = P\, x * \text{all } P\, xs'$, framing $P\, x * l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l')$ and then using our assumption on g.
3. As $\text{isList } l'\, xs' * l = \mathsf{inj}_2\, hd * hd \hookrightarrow (x, l') \Rightarrow \text{isList } l\, (x : xs')$ then it suffices by HT-CSQ to show

$$\{\text{isList } l\, (x : xs') * P\, x * I\, xs'\, v\}$$
$$f\, x\, v$$
$$\{r.\, \text{isList } l\, (x : xs) * I\, (x : xs')\, r\}$$

   which follows by framing $\text{isList } l\, (x : xs)$ and using our assumption on f.

Client: sumList

The following client is a function that computes the sum of a list of natural numbers by making a right-fold on +, 0 and the list. The code below is a slightly longer as it has to take into account that f takes the arguments as a pair.

$$\begin{aligned} \mathsf{sumList}\, l = \;& \mathsf{let}\, f = (\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y) \\ & \mathsf{in}\, \mathsf{foldr}(f, 0, l) \end{aligned}$$

The specification of the sumList function is as follows.

$$\forall l.\, \forall xs.\, \{\text{isList } l\, xs * \text{all isNat } xs\}\, \mathsf{sumList}\, l\, \{r.\, \text{isList } l\, xs * r = \Sigma_{x \in xs} x\}$$

where

$$\text{isNat } x \equiv \begin{cases} \mathsf{True} & \text{if } x \in \mathbb{N} \\ \mathsf{False} & \text{otherwise} \end{cases}$$

is a predicate stating that the argument is a natural number.

The specification of the sumList function states that given a list of natural numbers implemented by l, the result of $\mathsf{sumList}\, l$ is the sum of all the natural numbers in the list. The $\text{isList } l\, xs$ in the postcondition again ensures that sumList does not change the list.

[3] If we hadn't let f take the arguments as a tuple then we would have been stuck here as $f(x, -)$ is not an evaluation context. To get past this, we would need to change the specification, such that $f\, x$ would return a function g, that when applied to a' would satisfy the current specification for f *i.e.*, we would have to specify f using a nested Hoare triple and the specification for foldr would then involve a nested triple inside a nested triple.

Proof of the sumList specification Let l and xs be given. By HT-LET-DET it suffices to show

$$\begin{array}{l}\{\mathrm{isList}\ l\ xs * \mathrm{all}\,\mathrm{isNat}\ xs\}\\ \quad \mathsf{foldr}\,((\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y), 0, l)\\ \{r.\,\mathrm{isList}\ l\ xs * r = \Sigma_{x \in xs} x\}\end{array}$$

Using the specification for foldr, with isNat as P, $a' = \Sigma_{y \in ys} y$ as $I\, ys\, a'$, 0 as a, $(\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y)$ as f, l as l and xs as xs we get

$$\begin{array}{l}\left\{\begin{array}{l}\left(\forall x, a.\, \forall ys.\, \{\mathrm{isNat}\ x * a = \Sigma_{y \in ys} y\}\, (\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y)(x, a)\, \{r.r = \Sigma_{y \in (x:ys)}\}\right)\\ * \mathrm{isList}\ l\ xs * \mathrm{all}\,\mathrm{isNat}\ xs * 0 = \Sigma_{x \in []} x\end{array}\right\}\\ \quad \mathsf{foldr}\,((\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y), a, l)\\ \{r.\,\mathrm{isList}\ l\ xs * r = \Sigma_{x \in xs} x\}\end{array}$$

which is almost what we want. The difference being the precondition. By HT-CSQ it suffices to show

$$\begin{array}{l}\mathrm{isList}\ l\ xs * \mathrm{all}\,\mathrm{isNat}\ xs \Rightarrow \left(\begin{array}{c}\{\mathrm{isNat}\ x * a = \Sigma_{y \in ys} y\}\\ \forall x, a.\, \forall ys.\quad (\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y)(x, a)\\ \{r.r = \Sigma_{y \in (x:ys)}\}\end{array}\right)\\ \quad * \mathrm{isList}\ l\ xs * \mathrm{all}\,\mathrm{isNat}\ xs * 0 = \Sigma_{x \in []} x\end{array}$$

i.e., it is suffices to prove

1. $\forall x, a.\, \forall ys.\, \{\mathrm{isNat}\ \ x * a = \Sigma_{y \in ys} y\}\, (\lambda p, \mathsf{let}\, x = \pi_1\, p\, \mathsf{in}\, \mathsf{let}\, y = \pi_2\, p\, \mathsf{in}\, x + y)(x, a)\, \{r.r = \Sigma_{y \in (x:ys)}\}$
2. $0 = \Sigma_{x \in []} x$

without assuming anything.

The second item is immediate, and we leave the first as an exercise.

Exercise 4.40. Prove the specification 1 above. ◊

Client: filter

The following client implements a filter of some boolean predicate p on a list l, *i.e.*, it creates a *new list* whose elements are precisely those elements of l that satisfy p.

$$\begin{array}{lll}\mathsf{filter}(p, l) = & \mathsf{let}\, f = (\lambda y, & \mathsf{let}\, x = \pi_1\, y\, \mathsf{in}\\ & & \mathsf{let}\, xs = \pi_2\, y\, \mathsf{in}\\ & & \mathsf{if}\, p\, x\\ & & \mathsf{then}\, \mathsf{inj}_2\, (\mathsf{ref}(x, xs))\\ & & \mathsf{else}\, xs)\\ & \mathsf{in} & \\ & \mathsf{foldr}(f, \mathsf{inj}_1\, (), l) & \end{array}$$

Specification

$$\begin{array}{l}\forall P.\, \forall l.\, \forall xs.\, \{(\forall x.\, \{\mathsf{True}\}\, p\, x\, \{v.\, \mathrm{isBool}\ v * v = P\, x\}) * \mathrm{isList}\ l\ xs\}\\ \quad \mathsf{filter}(p, l)\\ \quad \{r.\, \mathrm{isList}\ l\ xs * \mathrm{isList}\ r\ (\mathrm{listFilter}\ P\ xs)\}\end{array}$$

where

$$\text{listFilter}\ P\,[] \equiv []$$
$$\text{listFilter}\ P\,(x : xs) \equiv \begin{cases} (x : (\text{listFilter}\ P\,xs)) & \text{if } P\,x = \mathsf{true} \\ \text{listFilter}\ P\,xs & \text{otherwise} \end{cases}$$

The specification

$$\forall x.\,\{\mathsf{true}\}\,p\,x\,\{v.\,\text{isBool}\ v * v = P\,x\}$$

in the precondition of the specification of filter states that p implements the boolean predicate P. The specification of filter states that given such an implementation p and a list l, whose elements corresponds to the mathematical sequence xs, then filter returns (without changing the original list) a list r whose elements are those that satisfy P.

Proof of the specification of filter Let P, l and xs be given. By Ht-let-det it suffices to show

$$\begin{array}{l}\{(\forall x.\,\{\mathsf{True}\}\,p\,x\,\{v.\,\text{isBool}\ v * v = P\,x\}) * \text{isList}\ l\,xs\} \\ \quad \mathsf{foldr}\,((\lambda y, \mathsf{let}\,x = \pi_1\,y\,\mathsf{in}\,\mathsf{let}\,xs = \pi_2\,y\,\mathsf{in}\,\mathsf{if}\,p\,x\,\mathsf{then}\,\mathsf{inj}_2\,(\mathsf{ref}(x, xs))\,\mathsf{else}\,xs), \mathsf{inj}_1\,(), l) \\ \{r.\,\text{isList}\ l\,xs * \text{isList}\ r\,(\text{listFilter}\ P\,xs)\}\end{array}$$

By applying the specification for foldr with True as $P\,x$, isList a (listFilter $P\,xs$) as $I\,xs\,a$,

$$(\lambda y, \mathsf{let}\,x = \pi_1\,y\,\mathsf{in}\,\mathsf{let}\,xs = \pi_2\,y\,\mathsf{in}\,\mathsf{if}\,p\,x\,\mathsf{then}\,\mathsf{inj}_2\,(\mathsf{ref}(x, xs))\,\mathsf{else}\,xs)$$

as f, xs as xs and l as l we get that

$$\begin{array}{l}\left\{\begin{array}{l}(\forall x'.\,\forall a'.\,\forall ys.\,\{\mathsf{True} * \text{isList}\ a'\ (\text{listFilter}\ P\,ys)\}\,p'\,(x', a')\,\{v.\,\text{isList}\ v\ (\text{listFilter}\ P\,(x' : ys))\}) \\ * \text{isList}\ l\,xs * \text{all}(\lambda x.\mathsf{True})\,xs * \text{isList}(\mathsf{inj}_1\,())\,(\text{listFilter}\ P\,[])\end{array}\right\} \\ \quad \mathsf{foldr}\,(p', \mathsf{inj}_1\,(), l) \\ \{r.\,\text{isList}\ l\,xs * \text{isList}\ r\,(\text{listFilter}\ P\,xs)\}\end{array}$$

where p' is a shorthand notation for

$$\lambda y, \mathsf{let}\,x = \pi_1\,y\,\mathsf{in}\,\mathsf{let}\,xs = \pi_2\,y\,\mathsf{in}\,\mathsf{if}\,p\,x\,\mathsf{then}\,\mathsf{inj}_2\,(\mathsf{ref}(x, xs))\,\mathsf{else}\,xs.$$

By Ht-csq we are done if we can show that

$$\begin{array}{l}(\forall x.\,\{\mathsf{True}\}\,p\,x\,\{v.\,\text{isBool}\ v * v = P\,x\}) * \text{isList}\ l\,xs \\ \Rightarrow \\ (\forall x'.\,\forall a'.\,\forall ys.\,\{\mathsf{True} * \text{isList}\ a'\ (\text{listFilter}\ P\,ys)\}\,p'\,(x', a')\,\{v.\,\text{isList}\ v\ (\text{listFilter}\ P\,(x' : ys))\}) \\ * \text{isList}\ l\,xs * \text{all}(\lambda x.\mathsf{True})\,xs * \text{isList}(\mathsf{inj}_1\,())\,(\text{listFilter}\ P\,[])\end{array}$$

Since isList $l\,xs$ clearly implies isList $l\,xs$ we only need to show.

1. $\forall x.\,\{\mathsf{True}\}\,p\,x\,\{v.\,\text{isBool}\ v * v = P\,x\}$
   $\Rightarrow \forall x'.\,\forall a'.\,\forall ys.\,\{\mathsf{True} * \text{isList}\ a'\ (\text{listFilter}\ P\,ys)\}\,p'\,(x', a')\,\{v.\,\text{isList}\ v\ (\text{listFilter}\ P\,(x' : ys))\}$
2. $\mathsf{True} \Rightarrow \text{all}(\lambda x.\mathsf{True})\,xs$
3. $\mathsf{True} \Rightarrow \text{isList}(\mathsf{inj}_1\,())\,(\text{listFilter}\ P\,[])$

Exercise 4.41. Prove the preceding three implications.
Hint: use induction for the first item. ◊

5 The later modality

The *later modality* is an essential feature of Iris. It will be used extensively in connection with invariants which we introduce in Section 7. However it can also be used for other things, chiefly for specifying and reasoning about higher-order programs using higher-order store. We show a more involved example of this in Section 7.8. In this section we introduce the later modality and its associated powerful Löb induction principle by a small, and rather artificial, example which has the benefit that it is relatively simple and only involves one new concept.

Recall that in untyped lambda calculus one can define a fixed-point combinator which can be used to express recursive functions. In our $\lambda_{\text{ref,conc}}$ language we have a primitive fixed point combinator $\mathsf{rec}\, f(x) = e$ but, for the purposes of introducing the later modality, we will show how to define a fixed-point combinator and prove a suitable specification for it.

To this end, we assume the following specification for λ terms.

$$\frac{S \vdash \{P\}\, e[v/x]\, \{u.Q\}}{S \vdash \{P\}\, (\lambda x.e)v\, \{u.Q\}}$$

Given a value F, the call-by-value Turing fixed-point combinator Θ_F is the following term

$$\Omega_F = \lambda r.F(\lambda x.rrx)$$
$$\Theta_F = \Omega_F \Omega_F$$

One can easily derive (exercise!), for any values F and v,

$$\Theta_F v \rightsquigarrow F(\lambda x.\Theta_F x)v$$

and thus, if $F = \lambda f x.e$ then one should think of Θ_F as $\mathsf{rec}\, f(x) = e$.

Now we wish to derive a useful proof rule for Θ_F, analogous to the simplified version of the rule Ht-Rec.[4]

$$\frac{\Gamma \mid S \land \forall v.\, \{P\}\, \Theta_F v\, \{u.Q\} \vdash \forall v.\, \{P\}\, F(\lambda x.\Theta_F x)v\, \{u.Q\}}{\Gamma \mid S \vdash \forall v.\, \{P\}\, \Theta_F v\, \{u.Q\}}\ \textsc{Ht-Turing-fp}$$

But at present there is nothing in the logic which would allow us to do so.

We are thus led to extend the logic with a new construct, the $\triangleright$ (pronounced "later") modality. The main punch of the later modality is the Löb rule:

$$\frac{Q \land \triangleright P \vdash P}{Q \vdash P}\ \textsc{Löb}$$

which is similar to a coinduction principle. It states that (in any context Q), if from $\triangleright P$ we can derive P, then we can also derive P without any assumptions. Note that the $\triangleright$ modality is necessary for the rule to be sound: if we admit the rule

$$\frac{Q \land P \vdash P}{Q \vdash P} \tag{10}$$

to the logic, then the logic would become inconsistent, in the sense that we could then prove $\mathsf{True} \vdash \mathsf{False}$.

[4] The rule is simplified by the removal of the "logical" variables y in the rule Ht-Rec. These are not essential for understanding this example and would only complicate the reasoning.

Exercise 5.1. Assuming (10) derive $\mathsf{True} \vdash \mathsf{False}$. ◊

Intuitively, the $\triangleright$ modality in the premise $\triangleright P$ in the Löb rule ensures that when proving P, we need to "do some work" before we can use P as an assumption.

The rest of the rules for the $\triangleright$ modality, listed in Figure 6, essentially ensure that that we can get the later modality at the correct place in order to use the Löb rule.

$$\textsc{Later-Mono}\ \frac{Q \vdash P}{\triangleright Q \vdash \triangleright P} \qquad \textsc{Later-weak}\ \frac{Q \vdash P}{Q \vdash \triangleright P} \qquad \textsc{Löb}\ \frac{Q \wedge \triangleright P \vdash P}{Q \vdash P} \qquad \triangleright\text{-}\exists\ \frac{\tau \text{ is inhabited} \qquad Q \vdash \triangleright \exists x : \tau. P}{Q \vdash \exists x : \tau. \triangleright P} \qquad \triangleright\text{-}\exists\ \frac{Q \vdash \exists x. \triangleright P}{Q \vdash \triangleright \exists x. P}$$

$$\textsc{Later-conj}\ \frac{R \vdash \triangleright(P \wedge Q)}{R \vdash \triangleright P \wedge \triangleright Q} \qquad \textsc{Later-disj}\ \frac{R \vdash \triangleright(P \vee Q)}{R \vdash \triangleright P \vee \triangleright Q} \qquad \textsc{Later-all}\ \frac{Q \vdash \triangleright \forall x. P}{Q \vdash \forall x. \triangleright P} \qquad \textsc{Later-sep}\ \frac{R \vdash \triangleright P * \triangleright Q}{R \vdash \triangleright(P * Q)}$$

Figure 6: Laws for the later modality. A type τ is *inhabited* if $\vdash t : \tau$ is derivable for some t.

Strengthening the Hoare rules In order for the later modality to be useful for proving specifications (Hoare triples) we will need to connect it in some way to the steps a program can take. Let us see how this comes up by trying to show Ht-Turing-fp.

We proceed by Löb induction so we assume $\triangleright \forall v. \{P\}\Theta_F v\{u.Q\}$ and we are to show

$$\forall v. \{P\}\Theta_F v\{u.Q\}.$$

Let v be a value. By using the Ht-bind and Ht-beta it suffices to show $\{P\}F(\lambda x.\Theta_F x)v\{u.Q\}$ and thus by the assumption of the rule we are proving it suffices to show

$$\forall v. \{P\}\Theta_F v\{u.Q\}.$$

However we only have the Löb induction hypothesis

$$\triangleright \forall v. \{P\}\Theta_F v\{u.Q\}$$

from which we cannot get what is needed. The issue is that we have not connected the later modality to the programming language constructs in any way. One way to do that is to have a stronger Ht-beta rule, which only assumes $\triangleright P$ in the precondition of the triple in the conclusion of the rule:

$$\textsc{Ht-beta}\ \frac{S \vdash \{P\}\, e[v/x]\, \{u.Q\}}{S \vdash \{\triangleright P\}\, (\lambda x.e)v\, \{u.Q\}}$$

The intuitive explanation for why this rule is sensible is that the term $(\lambda x.e)v$ takes one more step to evaluate than $e[v/x]$; hence any resources accessed by the body e will only be needed one step *later*.

Exercise 5.2. Convince yourself that the old beta rule is derivable from the new one using the rules presented thus far. ◊

The final piece we need is that if P is a persistent proposition (*e.g.*, a Hoare triple or equality) then $\triangleright P$ is also a persistent proposition, which implies that we can move it in and out of the preconditions (*c.f.* HT-HT).

Let us prove the rule HT-TURING-FP. We proceed by Löb induction so we assume

$$\triangleright \forall v.\, \{P\}\, \Theta_F\, v\, \{u.Q\}$$

and we are to show

$$\forall v.\, \{P\}\, \Theta_F\, v\, \{u.Q\}.$$

Let v be a value. By using LATER-WEAK and the rule of consequence HT-CSQ it suffices to show $\{\triangleright P\}\, \Theta_F\, v\, \{u.Q\}$. Since Hoare triples are persistent propositions this is equivalent to showing

$$\{\triangleright(\forall v.\, \{P\}\, \Theta_F\, v\, \{u.Q\} \wedge P)\}\, \Theta_F\, v\, \{u.Q\}$$

By the bind rule and the stronger rule HT-BETA introduced above it thus suffices to show

$$\{\forall v.\, \{P\}\, \Theta_F\, v\, \{u.Q\} \wedge P\}\, F(\lambda x.\Theta_F x) v\, \{u.Q\}$$

which again is equivalent (rule HT-HT) to showing $\{P\}\, F(\lambda x.\Theta_F x) v\, \{u.Q\}$ assuming $\forall v.\, \{P\}\, \Theta_F\, v\, \{u.Q\}$. But this is exactly the premise of the rule HT-TURING-FP, and thus the proof is concluded.

5.1 Stronger rules for Hoare triples

With the introduction of the later modality, we can strengthen the rules for Hoare triples, so that they allow the removal of the later modality in preconditions in those cases where the term we are specifying is not a value. From now on, we will use these stronger rules. We only list the rules which differ.

HT-LOAD

$$\frac{}{S \vdash \{\triangleright \ell \hookrightarrow u\}\, !\ell\, \{v.v = u \wedge \ell \hookrightarrow u\}}$$

HT-STORE

$$\frac{}{S \vdash \{\triangleright \exists u.\, \ell \hookrightarrow u\}\, \ell \leftarrow w\, \{v.v = () \wedge \ell \hookrightarrow w\}}$$

HT-REC

$$\frac{Q \vdash \{P\}\, e\, [(\mathsf{rec}\, f(x) = e)/f, v/x]\, \{\Phi\}}{Q \vdash \{\triangleright P\}\, (\mathsf{rec}\, f(x) = e) v\, \{\Phi\}}$$

HT-MATCH

$$\frac{S \vdash \{P\}\, e_i\, [u/x_i]\, \{v.Q\}}{S \vdash \{\triangleright P\}\, \mathsf{match}\, \mathsf{inj}_i\, u\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end}\, \{v.Q\}}$$

Exercise 5.3. Derive the rules in Section 4 apart from HT-REC, for which see Exercise 6.4 below, from the rules just listed. ◊

Exercise 5.4. This is a variant of Exercise 4.12 above deriving stronger rules for Hoare triples which allow us to remove more later modalities.

Show the following rules

$$\textsc{Ht-let}\quad \frac{S \vdash \{P\}\, e_1\, \{x.\triangleright Q\} \qquad S \vdash \forall v.\, \{Q[v/x]\}\, e_2\, [v/x]\, \{u.R\}}{S \vdash \{P\}\, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2\, \{u.R\}}$$

$$\textsc{Ht-let-det}\quad \frac{S \vdash \{P\}\, e_1\, \{x.\triangleright(x = v) \wedge \triangleright Q\} \qquad S \vdash \{Q[v/x]\}\, e_2\, [v/x]\, \{u.R\}}{S \vdash \{P\}\, \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2\, \{u.R\}}$$

$$\textsc{Ht-seq}\quad \frac{S \vdash \{P\}\, e_1\, \{\_.\triangleright Q\} \qquad S \vdash \{Q\}\, e_2\, \{u.R\}}{S \vdash \{P\}\, e_1; e_2\, \{u.R\}}\,. \qquad \textsc{Ht-If}\quad \frac{S \vdash \{P * v = \mathsf{true}\}\, e_2\, \{u.Q\} \qquad S \vdash \{P * v = \mathsf{false}\}\, e_3\, \{u.Q\}}{S \vdash \{\triangleright P\}\, \mathsf{if}\, v\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3\, \{u.Q\}}$$

◊

In the rest of the document we use these stronger variants of the rules. (For instance, when in future sections we refer to the sequencing rule, we refer to the one above involving a $\triangleright$.)

5.2 Recursively defined predicates

With the addition of the later modality we can extend the logic with *recursively defined predicates*. The terms of the logic are thus extended with

$$t ::= \cdots \mid \mu x : \tau.\, t$$

with the side-condition that the recursive occurrences must be *guarded*: in $\mu x.\, t$, the variable x can only appear under the later $\triangleright$ modality.

The typing rule for the new term is as expected

$$\frac{\Gamma, x : \tau \vdash t : \tau \qquad x \text{ is guarded in } t}{\Gamma \vdash \mu x : \tau.\, t : \tau}$$

where x is *guarded in* t if all of its occurrences in t are under the $\triangleright$ modality. For example, P is guarded in the following terms (where Q is a closed proposition)

$$\triangleright(P \wedge Q) \qquad\qquad (\triangleright P) \Rightarrow Q \qquad\qquad Q$$

but is *not* guarded in any of the following terms

$$P \wedge \triangleright Q \qquad\qquad P \Rightarrow Q \qquad\qquad P \vee Q.$$

We have a new rule for using guarded recursively defined predicates, which expresses the fixed-point property:

$$\textsc{Mu-fixed}\quad \frac{}{Q \vdash \mu x : \tau.\, t =_\tau t\,[\mu x : \tau.\, t/x]}$$

This rule is typically used in conjunction with Löb induction.

We shall see examples of the use of guarded recursively defined predicates later on.

6 The always modality

The rules HT-HT and HT-EQ and their generalizations involving $\triangleright$ used in Section 5 are somewhat ad-hoc. For instance, they do not allow us to move disjunctions of persistent propositions in and out of preconditions. In this section we present a more uniform way of treating such persistent propositions by way of a new modality $\Box$ (pronounced "always").

The intuitive reading of $\Box P$ is that it contains only those resources of P which can be split into a duplicable resource s in P and some other resource r. Thus $\Box P$ is like P except that it does not assert any exclusive ownership over resources. An example of a duplicable resource is the empty heap. If resources are heaps then that is the only duplicable resource. But later on will see more, and less trivial, examples of duplicable resources. Duplicable resources are important because we can always create a copy of them to give away to other threads. One way to state this precisely is the following rule

$$\Box P \dashv\vdash \Box P * P.$$

This rule is derivable from the axioms for the always modality, which are shown in Figure 7. We discuss these in the following.

We call a proposition P *persistent* if it satisfies $P \vdash \Box P$. Persistent propositions are important because they are duplicable, see Exercise 6.2.

An example of a persistent proposition is the equality relation. A *non-example* of a persistent proposition is the points-to predicate $x \hookrightarrow v$. Indeed, any heap in $x \hookrightarrow v$ contains at least one location, namely x. Hence the empty heap is not in $x \hookrightarrow v$. By the same reasoning we have $\Box(x \hookrightarrow v) \dashv\vdash \mathsf{False}$.

Given the above intuitive reading of $\Box P$, the first three axioms for $\Box$ in Figure 7, ALWAYS-MONO, ALWAYS-E and ALWAYS-IDEMP, are immediate. These three rules together allow us to prove the following "introduction" rule for the always modality.

$$\textsc{always-intro}\ \frac{\Box P \vdash Q}{\Box P \vdash \Box Q}\ .$$

Exercise 6.1. Prove the above introduction rule. ◊

The rules in the right column of Figure 7 state that True is persistent. If resources are heaps then True is the set of all heaps. Hence True in particular contains the empty heap and thus $\Box\mathsf{True} = \mathsf{True}$. The rest of the rules state that $\Box$ commutes with the propositional connectives. We do not explain why that is the case since that would require us to describe a model of the logic. (We describe the model of Iris in Section 12.)

Next comes the rule ALWAYS-SEP and the derived rule ALWAYS-SEP-DERIVED. They govern the interaction between separating conjunction, conjunction, and the always modality.

$$\textsc{Always-sep}\ \frac{S \vdash \Box P \wedge Q}{S \vdash \Box P * Q}$$

The intuitive reason for why this rule holds is that any resource r in $\Box P$ and Q can be split into a duplicable resource s in P (and since it is duplicable also in $\Box P$) and r. Hence r is in $\Box P * Q$.

There is an important derived rule which states that under the always modality conjunction and separating conjunction coincide. The rule is

$$\textsc{Always-sep-derived}\ \frac{S \vdash \Box(P \wedge Q)}{S \vdash \Box(P * Q)}$$

which is equivalent to the entailment

$$\Box(P \land Q) \vdash \Box(P * Q).$$

To derive this entailment we first show the following:

$$\frac{P \vdash \Box P}{P \vdash P * P}. \tag{11}$$

Indeed, using $P \vdash \Box P$ we have

$$P \vdash \Box P \vdash \Box P \land \Box P \vdash \Box P * \Box P \vdash P * P$$

using the rule Always-sep in the third step.

Next, by the fact that $\Box$ commutes with conjunction and (11), we have

$$\Box P \land \Box Q \vdash \Box(P \land Q) \vdash \Box(P \land Q) * \Box(P \land Q)$$

Now, using the fact that $P \land Q \vdash P$ and $P \land Q \vdash Q$ and monotonicity of $\Box$ and $*$, we moreover have

$$\Box(P \land Q) * \Box(P \land Q) \vdash \Box P * \Box Q.$$

Hence we have proved

$$\Box P \land \Box Q \vdash \Box P * \Box Q. \tag{12}$$

With this we can finally derive Always-sep-derived as follows

$$\Box(P \land Q) \vdash \Box\Box(P \land Q) \vdash \Box(\Box P \land \Box Q) \vdash \Box(\Box P * \Box Q) \vdash \Box(P * Q)$$

where in the last step we use $\Box P \vdash P$ for any P and monotonicity of separating conjunction.

Exercise 6.2. Using similar reasoning show the following derived rules.

1. $\Box\Box P \vdash \Box P$
2. $\Box(P \Rightarrow Q) \vdash \Box P \Rightarrow \Box Q$
3. $P \Rightarrow Q \vdash P \mathrel{-\!\!*} Q$
4. $\Box(P \mathrel{-\!\!*} Q) \vdash \Box(P \Rightarrow Q)$
5. $\Box(P \mathrel{-\!\!*} Q) \vdash \Box P \mathrel{-\!\!*} \Box Q$
6. $(P \mathrel{-\!\!*} (Q * \Box R)) * P \vdash (P \mathrel{-\!\!*} (Q * \Box R)) * P * \Box R$

The last item is often useful. It states that we can get persistent propositions out of $\mathrel{-\!\!*}$ without consuming any resources. ◊

Exercise 6.3. Derive $\Box(x \hookrightarrow v) \vdash \mathsf{False}$ using the rules in Figure 7 and the basic axioms of the points-to predicate. ◊

Exercise 6.4. Show the following derived rule for recursive function calls from the rule Ht-Rec above.

$$\frac{\Gamma, f : \mathit{Val} \mid Q \land \forall y.\, \forall v.\, \{\triangleright P\}\, f\, v\, \{\Phi\} \vdash \forall y.\, \forall v.\, \{P\}\, e[v/x]\, \{\Phi\}}{\Gamma \mid Q \vdash \forall y.\, \forall v.\, \{\triangleright P\}\, (\mathsf{rec}\, f(x) = e) v\, \{\Phi\}}$$

Hint: Start with Löb induction. Use the rule Ht-always to move the Löb induction hypothesis into the precondition. Then use the recursion rule Ht-Rec, and again Ht-always. You are almost there.

◊

$$\textsc{always-mono}\ \frac{P \vdash Q}{\Box P \vdash \Box Q} \qquad \textsc{always-E}\ \frac{}{\Box P \vdash P} \qquad \textsc{always-idemp}\ \frac{}{\Box P \vdash \Box\Box P}$$

$$\begin{array}{rcl} \mathsf{True} & \dashv\vdash & \Box\,\mathsf{True} \\ \Box(P \wedge Q) & \dashv\vdash & \Box P \wedge \Box Q \\ \Box(P \vee Q) & \dashv\vdash & \Box P \vee \Box Q \\ \Box \triangleright P & \dashv\vdash & \triangleright \Box P \\ \forall x.\, \Box P & \dashv\vdash & \Box \forall x.\, P \\ \Box \exists x.\, P & \dashv\vdash & \exists x.\, \Box P \end{array}$$

$\Box$ governs the interaction between conjunction and separating conjunction.

$$\textsc{Always-sep}\ \frac{S \vdash \Box P \wedge Q}{S \vdash \Box P * Q}$$

Finally we have two kinds of primitive persistent propositions.

$$t =_\tau t' \dashv\vdash \Box(t =_\tau t') \qquad\qquad \{P\}\, e\, \{\Phi\} \dashv\vdash \Box \{P\}\, e\, \{\Phi\}$$

We have the following rule Ht-always which, combined with the rules above, generalizes Ht-Ht, Ht-Eq and Ht-False.

$$\textsc{Ht-always}\ \frac{\Box Q \wedge S \vdash \{P\}\, e\, \{v.\, R\}}{S \vdash \{P \wedge \Box Q\}\, e\, \{v.\, R\}}$$

Figure 7: Laws for the always modality.

7 Invariants and ghost state

In this section we finally extend the logic to support reasoning about concurrent programs. Two crucial ingredients in achieving this are *invariants* and *ghost state*. Invariants are used to allow different threads to access the same resources, but in a controlled way, and ghost state is used to keep track of additional information, e.g., relationships between different program variables, needed to verify the program. Although ghost state is already useful in a sequential setting, it becomes much more powerful and expressive in connection with invariants.

First we introduce the parallel composition construct par with the help of which we will motivate and introduce invariants and ghost state. Later on we will provide a rule for the fork primitive and derive the rule for par rule from it. This derivation however requires more advanced concepts so we postpone it until the next section. The main reason we use par instead of fork in this section is that it is easier to use in the small motivating examples, since it is a higher-level construct.

7.1 The par construct

Using the fork primitive we can implement a "par" construct which takes two expressions e_1 and e_2, runs them in parallel, waits until both finish, and then returns a pair of values to which e_1 and e_2 evaluated. First we have the auxiliary methods spawn and join. We use syntactic sugar None for $\mathsf{inj}_1\,()$ and Some x for $\mathsf{inj}_2\,x$.

The method spawn takes a function and runs it in another thread, but it also allocates a reference cell. This cell will hold the result of running the given function. The method join waits until the value at the given location is not None.

$$\begin{aligned}\mathsf{spawn} &:= \lambda f.\mathsf{let}\,c = \mathsf{ref}(\text{None})\,\mathsf{in}\,\mathsf{fork}\,(c \leftarrow \text{Some}(f\,()))\,;c\\ \mathsf{join} &:= \mathsf{rec}\,f(c) = \mathsf{match}\,!c\,\mathsf{with}\\ &\qquad\qquad \text{Some}\,x \Rightarrow x\\ &\qquad\qquad \mid \text{None} \quad \Rightarrow f(c)\\ &\qquad\qquad \mathsf{end}\end{aligned}$$

Using these terms we can implement par as follows.

$$\begin{aligned}\mathsf{par} := \lambda f_1\,f_2.&\mathsf{let}\,h = \mathsf{spawn}\,f_1\,\mathsf{in}\\ &\mathsf{let}\,v_2 = f_2\,()\,\mathsf{in}\\ &\mathsf{let}\,v_1 = \mathsf{join}(h)\,\mathsf{in}\\ &(v_1, v_2)\end{aligned}$$

Finally, we define new infix notation for the par construct. It wraps the given expressions into thunks:

$$e_1 \parallel e_2 := \mathsf{par}(\lambda\_.e_1)(\lambda\_.e_2)$$

In words the functions do the following. We spawn a new thread and run the function f_1 there. In the current thread we execute f_2 and then wait until f_1 finishes (the call to join). The notation $e_1 \parallel e_2$ is then a wrapper around par. We need to make thunks out of expressions because our language is call-by-value. If we were to pass expression e_1 and e_2 directly to par, then they

would be evaluated in the current thread *before* being passed to spawn, hence defeating the purpose of par.

The $\|$ construct satisfies the following specification.

$$\textsc{Ht-par}\quad \frac{S \vdash \{P_1\}\, e_1\, \{v.Q_1\} \qquad S \vdash \{P_2\}\, e_2\, \{v.Q_2\}}{S \vdash \{P_1 * P_2\}\, e_1 \parallel e_2\, \{v.\exists v_1\, v_2.\, v = (v_1, v_2) * Q_1[v_1/v] * Q_2[v_2/v]\}}$$

The rule states that we can run e_1 and e_2 in parallel if they have *disjoint* footprints and that in this case we can verify the two components separately. Thus this rule is sometimes also referred to as the *disjoint concurrency rule*.

With the concepts introduced so far we can verify simple examples of concurrent programs. Namely those where threads do not communicate. We hasten to point out that there are many important examples of such programs. For instance, list sorting algorithms such as quick sort and merge sort, where the recursive calls operate on disjoint lists of elements.

Exercise 7.1. Prove the following specification.

$$\{\ell_1 \hookrightarrow n * \ell_2 \hookrightarrow m\}\, (e_1 \parallel e_2); !\ell_1 + !\ell_2\, \{v.v = n + m + 2\}$$

where, for $i \in \{1, 2\}$, e_i is the program $\ell_i \leftarrow !\ell_i + 1$. ◊

Exercise 7.2. Consider the following implementation of ◊

However the Ht-par rule does not suffice to verify any concurrent programs which modify a shared location. For instance, we cannot use it to prove

$$\{\ell \hookrightarrow n\}\, (e \parallel e); !\ell\, \{v.v \geq n\} \tag{13}$$

where e is the program $\ell \leftarrow !\ell + 1$. The problem here is that we cannot split the $\ell \hookrightarrow n$ predicate to give to the two subcomputations.

Note that we cannot hope to prove

$$\{\ell \hookrightarrow n\}\, (e \parallel e); !\ell\, \{v.v = n + 2\}$$

since the command $\ell \leftarrow !\ell + 1$ is not atomic: both threads could first read the value stored at ℓ, which is n, and then write back the value $n + 1$.

The best we can hope to prove is

$$\{\ell \hookrightarrow n\}\, (e \parallel e); !\ell\, \{v.v = n + 1 \vee v = n + 2\} \tag{14}$$

However this specification is considerably harder to prove than (13). To avoid having to introduce too many concepts at once, we first focus on describing the necessary concepts for proving (13). We return to proving the specification (14) in Example 7.31 after we introduce the necessary concepts.

What we need is the ability to *share* the predicate $\ell \hookrightarrow n$ among the two threads running in parallel. *Invariants* enable such sharing: they are persistent resources,n hence duplicable, hence sharable among several threads.

7.2 Invariants

To introduce invariants we need to add a type of invariant names InvName to the logic. Invariants are associated with names and names are used to ensure that we do not open an invariant more than once. We explain why this is needed later on.

We add a new term $\boxed{P}^{\iota}$ to the logic, which should be read as invariant P *named* ι, or associated with the name ι.

The typing rule for the new construct is as follows.

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash \iota : \mathsf{InvName}}{\Gamma \vdash \boxed{P}^{\iota} : \mathsf{Prop}}$$

That is, we can make an invariant out of any proposition and any name. Notice in particular that we can form *nested invariants*, *e.g.*, terms of the form $\boxed{\boxed{P}^{\iota}}^{\iota'}$.

The rules for invariants are listed in Figure 8 on page 50. As mentioned above we need to make sure that we do not open the same invariant more than once (see Example 7.4 for an example of what goes wrong if we allow opening an invariant twice). For this reason we need to annotate Hoare triples with an infinite set of invariant names $\mathcal{E}$. This set identifies the invariants we are allowed to use; see the rule Ht-inv-open. If there is no annotation on the Hoare triple then $\mathcal{E} = \mathsf{InvName}$, the set of all invariant names. With this convention all the previous rules are still valid.

With the addition of invariant names to Hoare triples there is a need to relate Hoare triples with different sets of invariant names. We just have one rule for that:

$$\text{Ht-mask-weaken}\quad \frac{S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}_1} \qquad \mathcal{E}_1 \subseteq \mathcal{E}_2}{S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}_2}}$$

This weakening rule allows us to add more invariant names. Intuitively it is sound, because if we are *allowed* to use more invariants then surely we can prove more specifications.

We now explain the rules for invariants.

Invariants are persistent The essential property of invariants is that they can be shared by different threads. The precise way to state this property in Iris is that invariants are persistent (the rule Inv-persistent).

Allocating invariants The invariant allocation rule

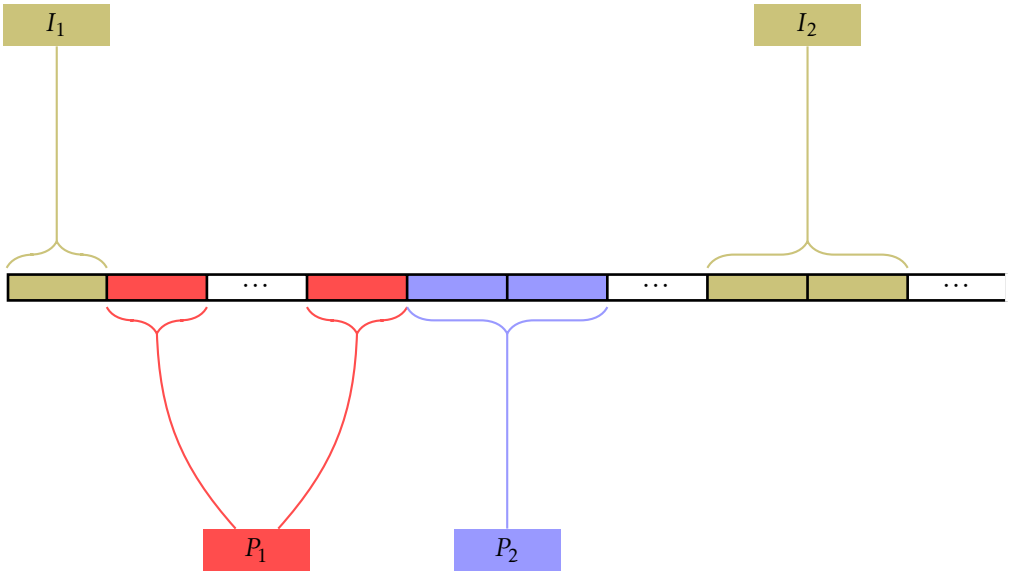
$$\text{Ht-inv-alloc}\quad \frac{\mathcal{E} \subseteq \mathsf{InvName} \qquad \mathcal{E}\ \text{infinite} \qquad S \wedge \exists \iota \in \mathcal{E}.\, \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E}}}{S \vdash \{\triangleright P * Q\}\, e\, \{v.R\}_{\mathcal{E}}}$$

has the following interpretation. To verify a program e, which will typically contain either fork or parallel composition $\|$, we want to share the resources described by P between different threads. To this ends we give away the resources to an invariant, *i.e.*, we lose the resources P, but we obtain an invariant $\boxed{P}^{\iota}$ for *some* ι. We can only specify that ι comes from some infinite set of names, but no more. The ability to choose $\mathcal{E}$ is needed when we wish to use multiple invariants. We want different invariants to be named differently so that we can open multiple invariants at the same time; cf. the explanation of the Ht-inv-open rule below.

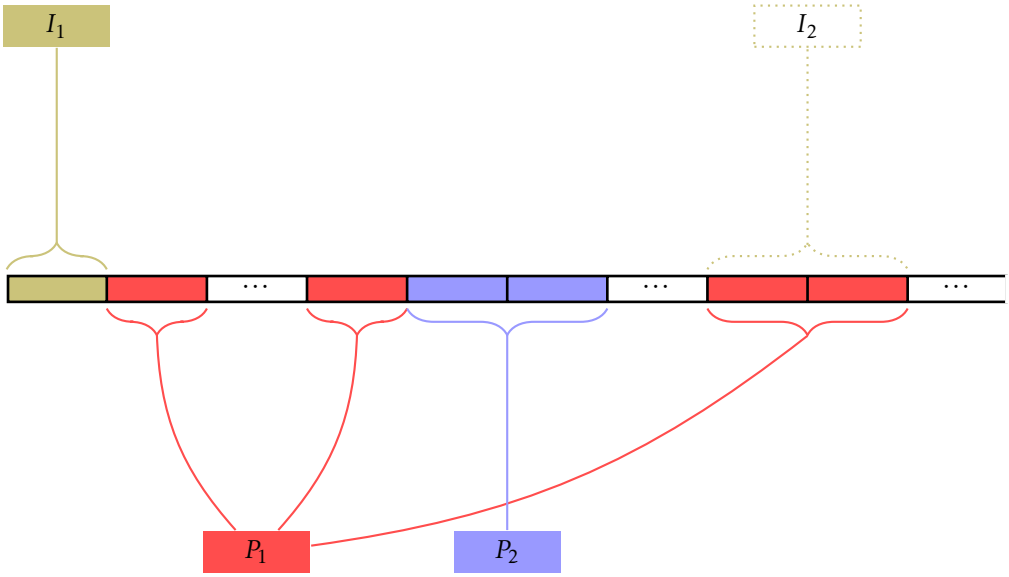
Invariants are persistent, so giving away resources to invariants is not without cost. The cost is that invariants can only be used in a restricted way, namely by the invariant opening rule.

Footprint reading of Hoare triples With the introduction of invariants, the "minimal footprint" reading of Hoare triples mentioned in Section 4 must be refined. Now the resources needed to run the program e can either be in the precondition P of the triple $\{P\}e\{v.Q\}$ or they can be governed by one or more invariants. Thus we will often prove triples of the form $\{\mathsf{True}\}e\{v.Q\}$, for some Q, where e accesses shared state governed by an invariant. See Example 7.5, in particular the proof of the triple (15) on page 51.

Graphically, we can depict the situation as follows.



The heap (and other resources) is split between local state owned by the two threads (resources P_1 owned by the first thread, and resources P_2 owned by the second thread) and some shared stated owned by invariants (in this case I_1 and I_2). Individual threads can access the state owned by invariants and temporarily transfer it to their local state, using the invariant opening rule we will see below. Thus if, for instance, the first thread opens invariant I_2, we can depict the state as follows.



The resources owned by the invariant are temporarily transferred to the local state of the first thread. This is the essence of the invariant opening rule.

Using invariants The invariant opening rule

$$\textsc{Ht-inv-open}\quad \frac{e \text{ is an atomic expression} \qquad S \wedge \boxed{P}^{\iota} \vdash \{\triangleright P * Q\}\, e\, \{v. \triangleright P * R\}_{\mathcal{E}}}{S \wedge \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E} \uplus \{\iota\}}}$$

is the only way to get access to the resources governed by an invariant. The rule states that if we know an invariant $\boxed{P}^{\iota}$ exists, we can *temporarily*, for one atomic step, get access to the resources. This rule is the reason we need to annotate the Hoare triples with sets of invariant names $\mathcal{E}$. This set of names contains names of those invariants which we are allowed to open and we refer to $\mathcal{E}$ as a *mask*. In particular, we cannot open the same invariant twice (see Example 7.4 for an example of what goes wrong if we allow opening an invariant twice).

The restriction of the term e to be an *atomic* expression is also essential; see Example 7.7. An expression e is *atomic* if it steps to a value in a single execution step.

Existing Hoare triple rules The existing rules for Hoare triples, *e.g.*, those in Figure 5, are all still valid with arbitrary masks, but the premises and conclusions of the rules must be annotated with the same mask. For example, the rule Ht-beta becomes

$$\textsc{Ht-beta}\quad \frac{S \vdash \{P\}\, e\, [v/x]\, \{u.Q\}_{\mathcal{E}}}{S \vdash \{\triangleright P\}\, (\lambda x.e) v\, \{u.Q\}_{\mathcal{E}}}$$

for an arbitrary invariant mask $\mathcal{E}$.

The rules in Figure 8 will be considerably *generalised* and *simplified* later, but for that we need concepts we have not yet introduced.

Before proceeding with an example we need one more rule, namely a stronger frame rule, which is only applicable in certain cases. The rule is needed because opening invariants only

gives access to the resources *later*. This is essential. The logic would be inconsistent otherwise, though proof of this fact is not yet within our reach – we will see why in Section 11.

The stronger frame rule is the following

$$\textsc{Ht-frame-atomic}\quad \frac{e \text{ is an atomic expression} \qquad S \vdash \{P\}\, e\, \{v.Q\}}{S \vdash \{P * \triangleright R\}\, e\, \{v.Q * R\}}$$

This rule is useful because typically an invariant will contain something akin to $\ell \hookrightarrow v$, plus some additional facts about v, and the expression e will be either reading from or writing to the location ℓ. This rule, together with the invariant opening rule, allows us to get the facts about v *now* (note that there is no $\triangleright$ on R in the postcondition) after reading the value.

Exercise 7.3 (Later False)**.** We will often use an invariant to tell us that some cases are impossible. That is, we have some resources *now* that are incompatible with those held by the invariant. But the invariant gives us those resources, and hence the inconsistency, *later*. Using Ht-frame-atomic show the following triples.

$$\{\triangleright(\mathsf{False})\}\, \ell \leftarrow v\, \{v.Q\} \qquad \{\triangleright(\mathsf{False})\}\, !\ell\, \{v.Q\} \qquad \{\triangleright(\mathsf{False})\}\, \mathsf{ref}(v)\, \{v.Q\}$$

$$\{\triangleright(\mathsf{False})\}\, \mathsf{cas}(\ell, v_1, v_2)\, \{v.Q\}$$

and use them to derive the following rule.

$$\textsc{Ht-later-false}\quad \frac{e \text{ is an atomic expression}}{\{\triangleright(\mathsf{False})\}\, e\, \{v.Q\}}$$

Hint: use Ht-csq with the fact that False entails anything and $\triangleright$ is monotone (the rule Later-Mono). ◊

$$\textsc{Inv-persistent}\quad \frac{}{\boxed{P}^{\iota} \vdash \Box \boxed{P}^{\iota}} \qquad \textsc{Ht-inv-alloc}\quad \frac{\mathcal{E} \text{ infinite} \qquad S \wedge \exists \iota \in \mathcal{E}.\, \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E}}}{S \vdash \{\triangleright P * Q\}\, e\, \{v.R\}_{\mathcal{E}}}$$

$$\textsc{Ht-inv-open}\quad \frac{e \text{ is an atomic expression} \qquad S \wedge \boxed{P}^{\iota} \vdash \{\triangleright P * Q\}\, e\, \{v.\triangleright P * R\}_{\mathcal{E}}}{S \wedge \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E} \uplus \{\iota\}}}$$

Figure 8: Rules for invariants.

Example 7.4 (Opening an invariant twice leads to an inconsistency)**.** This example demonstrates a problem with opening an invariant more than once. Suppose the invariant opening rule Ht-inv-open did not remove the name ι from the possible set of invariants to open, *i.e.*, suppose the rule was instead

$$\frac{e \text{ is an atomic expression} \qquad S \wedge \boxed{P}^{\iota} \vdash \{\triangleright P * Q\}\, e\, \{v.\triangleright P * R\}_{\mathcal{E} \uplus \{\iota\}}}{S \wedge \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E} \uplus \{\iota\}}}$$

Then we can derive the following nonsensical triple.

$$\{\ell \hookrightarrow 0\}\,!\ell\,\{v.v = 3\}.$$

Indeed, using the invariant allocation rule HT-INV-ALLOC we just need to prove

$$\exists \iota \in \mathsf{InvName}.\, \boxed{\ell \hookrightarrow 0}^{\iota} \vdash \{\mathsf{True}\}\,!\ell\,\{v.v = 3\}.$$

Opening the invariant once we have to show

$$\boxed{\ell \hookrightarrow 0}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow 0)\}\,!\ell\,\{v.v = 3 \wedge \triangleright(\ell \hookrightarrow 0)\}.$$

And opening again we need to show

$$\boxed{\ell \hookrightarrow 0}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow 0 * \ell \hookrightarrow 0)\}\,!\ell\,\{v.v = 3 \wedge \triangleright(\ell \hookrightarrow 0 * \ell \hookrightarrow 0)\}.$$

Since $\ell \hookrightarrow 0 * \ell \hookrightarrow 0$ is equivalent to False we can use HT-LATER-FALSE to prove the triple.

Hence opening the same invariant twice cannot be allowed. ■

Example 7.5. We now have sufficient rules to prove specification (13) from page 46.

$$\{\ell \hookrightarrow n\}\,(e \parallel e);!\ell\,\{v.v \geq n\}.$$

We start off by allocating an invariant. One might first guess that the invariant is $\ell \hookrightarrow n$. However this does not work since the value at location ℓ does in fact change, so is not *invariant*. Technically, we can see that, to use the invariant opening rule, we need to reestablish the invariant (the $\triangleright P$ in the post-condition).

Instead, we use the weaker predicate $I = \exists m.\, m \geq n \wedge \ell \hookrightarrow m$, which is an invariant. To show (13) we first allocate the invariant I using HT-INV-ALLOC. This we can do by the rule of consequence HT-CSQ since $\ell \hookrightarrow n$ implies I, and so also $\triangleright I$.

Thus we have to prove

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,(e \parallel e);!\ell\,\{v.v \geq n\} \tag{15}$$

for some ι.

Using the derived sequencing rule HT-SEQ we need to show the following two triples

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,(e \parallel e)\,\{\_.\mathsf{True}\}.$$
$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,!\ell\,\{v.v \geq n+1\}.$$

We show the first one; during the proof of that we will need to show the second triple as well. Using the rule HT-PAR, the proof of the first triple reduces to showing

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,e\,\{\_.\mathsf{True}\}$$

where, recall, e is the term $\ell \leftarrow !\ell + 1$. Note that we cannot open the invariant now since the expression e is not atomic.

Using the bind rule we first show

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,!\ell\,\{v.v \geq n\}.$$

Note that this is exactly the second premise of the sequencing rule mentioned above. To show this triple, we use the invariant opening rule Ht-inv-open, and thus it remains to show

$$\{\triangleright I\}\,!\ell\,\{v.v \geq n \wedge \triangleright I\}_{\mathsf{InvName}\setminus\{\iota\}}.$$

Using the rule Ht-frame-atomic together with the rule Ht-load and structural rules we have

$$\{\triangleright I\}\,!\ell\,\{v.v = m \wedge m \geq n \wedge \ell \hookrightarrow m\}_{\mathsf{InvName}\setminus\{\iota\}}.$$

From this we easily derive the needed triple.

To show the second premise of the bind rule we need to show

$$\boxed{I}^{\iota} \vdash \forall m.\,\{m \geq n\}\,\ell \leftarrow (m+1)\,\{\_.\mathsf{True}\}.$$

To show this we again use the invariant opening rule and Ht-frame-atomic.

Exercise 7.6. Show this claimed specification in detail. ◊

This concludes the proof. ■

Example 7.7 (Restriction to atomic expressions in Ht-inv-open is necessary.)**.** The restriction on atomic expressions in the invariant opening rule is necessary. Consider the following program, call it e

$$(\ell \leftarrow 4; \ell \leftarrow 3) \,\|\, !\ell$$

and the invariant $I = \ell \hookrightarrow 3$. Suppose the rule Ht-inv-open did not restrict expressions e to be atomic. Then we could allocate the invariant $\boxed{I}^{\iota}$ and then use the rule Ht-par. Without the atomicity restriction it is easy to show (exercise!)

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,\ell \leftarrow 4; \ell \leftarrow 3\,\{\_.\mathsf{True}\}$$

and

$$\boxed{I}^{\iota} \vdash \{\mathsf{True}\}\,!\ell\,\{v.v = 3\}$$

Hence, by Ht-par, we conclude

$$\{\ell \hookrightarrow 3\}\,e\,\{v.v = ((), 3)\}.$$

However, the pair $((), 4)$ is also a possible result of executing e (the second thread could read ℓ just after it was set to 4 by the first thread). Thus the logic would not be sound with respect to the operational behaviour of the programming language. ■

7.3 A peek at ghost state

The specification (13) is weaker than what happens operationally. The following specification

$$\{\ell \hookrightarrow n\}\,(e \,\|\, e); !\ell\,\{v.v \geq n+1\} \tag{16}$$

where e is again the program $\ell \leftarrow !\ell + 1$ is sound. However the logic we have introduced thus far does not allow us to prove it. Invariants allow us to make resources available to different threads, but exactly because they are shared by different threads, the resources governed by

them need to be preserved, *i.e.*, the invariant has to be reestablished after each step of execution. Thus, for instance, when the invariant is $\ell \hookrightarrow n$ the location ℓ must always point to the value n.

We could allow the state to change by using an invariant such as $\ell \hookrightarrow n \vee \ell \hookrightarrow (n{+}1)$. However with the concepts introduced until now we cannot have an invariant that would ensure that once $\ell \hookrightarrow (n+1)$ holds, the location ℓ will never point to n again.

One way to express this is using *ghost state*. In Iris, ghost state is an additional kind of primitive resource, analogous to the points-to predicate. Other names for the same concept are *auxiliary state*, *logical state*, or *abstract state*, to contrast it with *concrete state*, which is the concrete program configuation, *i.e.*, a heap and threadpool.

Iris supports a uniform treatment of ghost state, but in this subsection we start out more concretely, with just enough ghost state to prove (16).

To work with ghost state we extend Iris with a new type GhostName of *ghost names*, which we typically write as γ. Ghost names are to be thought of as analoguos to concrete locations in the heap, but for the abstract state of the program. Hence ghost names are also sometimes referred to as ghost variables. There are no special operations on ghost names. Ghost names can only be introduced by ghost name allocation, which we explain below.

To prove (16) we need two additional primitive resource propositions, indexed by GhostName: $\boxed{\mathrm{S}}^{\gamma}$ and $\boxed{\mathrm{F}}^{\gamma}$. These satisfy the following basic properties:

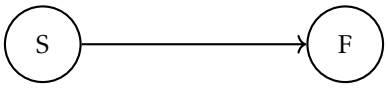
F-DUPLICABLE
$$\frac{}{\boxed{\mathrm{F}}^{\gamma} \vdash \boxed{\mathrm{F}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma}}$$

S-S-INCOMPATIBLE
$$\frac{}{\boxed{\mathrm{S}}^{\gamma} * \boxed{\mathrm{S}}^{\gamma} \vdash \mathsf{False}}$$

S-F-INCOMPATIBLE
$$\frac{}{\boxed{\mathrm{S}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma} \vdash \mathsf{False}}$$

The way to think about these propositions is that $\boxed{\mathrm{S}}^{\gamma}$ is the "start" token. The invariant will start out in this "state". The proposition $\boxed{\mathrm{F}}^{\gamma}$ is the "finished" token. Once the invariant is in this state, it can never go back to the state $\boxed{\mathrm{S}}^{\gamma}$.

Conceptually, the tokens are used to encode the following transition system.



Additionally, we need rules relating these tokens to Hoare triples:

HT-TOKEN-ALLOC
$$\frac{T \in \{\mathrm{S}, \mathrm{F}\} \qquad S \vdash \left\{\exists \gamma. \boxed{T}^{\gamma} * P\right\} e \left\{v. Q\right\}}{S \vdash \{P\}\, e\, \{v. Q\}}$$

HT-TOKEN-UPDATE-PRE
$$\frac{S \vdash \left\{\boxed{\mathrm{F}}^{\gamma} * P\right\} e \left\{v. Q\right\}}{S \vdash \left\{\boxed{\mathrm{S}}^{\gamma} * P\right\} e \left\{v. Q\right\}}$$

HT-TOKEN-UPDATE-POST
$$\frac{S \vdash \{P\}\, e \left\{v. \boxed{\mathrm{S}}^{\gamma} * Q\right\}}{S \vdash \{P\}\, e \left\{v. \boxed{\mathrm{F}}^{\gamma} * Q\right\}}$$

Using these rules, we now prove (16). The invariant we pick is the following predicate, parametrised by $\gamma \in \mathsf{GhostName}$.

$$I(\gamma) = \exists m. \ell \hookrightarrow m * \left(\left(\boxed{\mathrm{S}}^{\gamma} \wedge m \geq n\right) \vee \left(\boxed{\mathrm{F}}^{\gamma} \wedge m \geq (n+1)\right)\right)$$

The idea is as explained above. The invariant can be in two "states". It will be allocated in the first state, with the "start" token S, since we know that the current value stored at ℓ is at least n. Then, when a thread increases the value stored at ℓ, we will update the invariant, so that it is in the "finished" state. This pattern of using special ghost state tokens and disjunction to encode information about the execution of the program in the invariant is typical, and we shall see more of it later.

Example 7.8. So, let us start proving. We start off by using the rule HT-TOKEN-ALLOC plus HT-EXIST, so we have to prove

$$\left\{\boxed{\mathrm{S}}^{\gamma} * \ell \hookrightarrow n\right\}(e \parallel e);!\,\ell\,\{v.v \geq n+1\}.$$

We then again use the sequencing rule HT-SEQ, but this time the intermediate proposition is not True, but $\boxed{\mathrm{F}}^{\gamma}$, *i.e.*, we prove the following two triples

$$\left\{\boxed{\mathrm{S}}^{\gamma} * \ell \hookrightarrow n\right\} e \parallel e \left\{v.\boxed{\mathrm{F}}^{\gamma}\right\} \tag{17}$$

$$\left\{\boxed{\mathrm{F}}^{\gamma}\right\} !\,\ell\,\{v.v \geq n\}. \tag{18}$$

We begin by showing the first triple. Start by using the invariant allocation rule HT-INV-ALLOC. This is allowed by an application of the rule of consequence HT-CSQ since $\boxed{\mathrm{S}}^{\gamma} * \ell \hookrightarrow n$ implies $I(\gamma)$. Hence we have to prove

$$\boxed{I(\gamma)}^{\iota} \vdash \{\mathsf{True}\}\, e \parallel e \left\{v.\boxed{\mathrm{F}}^{\gamma}\right\}.$$

Since $\boxed{\mathrm{F}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma}$ implies $\boxed{\mathrm{F}}^{\gamma}$ it suffices (by the rule of consequence) to use the parallel composition rule HT-PAR and prove

$$\boxed{I(\gamma)}^{\iota} \vdash \{\mathsf{True}\}\, e \left\{v.\boxed{\mathrm{F}}^{\gamma}\right\}.$$

Again, using the bind rule, we first need to prove

$$\boxed{I(\gamma)}^{\iota} \vdash \{\mathsf{True}\}\, !\,\ell\,\{v.v \geq n\}.$$

Exercise!

For the other premise of the bind rule, we now have to show

$$\boxed{I(\gamma)}^{\iota} \vdash \{m \geq n\}\, \ell \leftarrow (m+1) \left\{\_.\boxed{\mathrm{F}}^{\gamma}\right\}.$$

To open the invariant we need an atomic expression. We use the rule HT-BIND-DET to evaluate $m+1$ to a value using the rule HT-OP. We then open the invariant and after using the rule HT-DISJ and other structural rules we need to prove the following two triples

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\triangleright(\ell \hookrightarrow m * \boxed{\mathrm{S}}^{\gamma} \wedge m \geq n)\right\} \ell \leftarrow (m+1) \left\{\_.\triangleright I(\gamma) * \boxed{\mathrm{F}}^{\gamma}\right\}$$

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\triangleright(\ell \hookrightarrow m * \boxed{\mathrm{F}}^{\gamma} \wedge m \geq (n+1))\right\} \ell \leftarrow (m+1) \left\{\_.\triangleright I(\gamma) * \boxed{\mathrm{F}}^{\gamma}\right\}$$

We only show the first one, and leave the second one as an exercise. We note however that duplicability of $\boxed{\mathrm{F}}^{\gamma}$ is essential.

Using the rules HT-FRAME-ATOMIC and HT-STORE we derive the following entailment.

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\triangleright(\ell \hookrightarrow m * \boxed{\mathrm{S}}^{\gamma} \wedge m \geq n)\right\} \ell \leftarrow (m+1) \left\{v.v = () \wedge \ell \hookrightarrow (m+1) * \boxed{\mathrm{S}}^{\gamma} \wedge m \geq n\right\}$$

Following up with HT-TOKEN-UPDATE-POST we get

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\triangleright(\ell \hookrightarrow m * \boxed{\mathrm{S}}^{\gamma} \wedge m \geq n)\right\} \ell \leftarrow (m+1) \left\{v.v = () \wedge \ell \hookrightarrow (m+1) * \boxed{\mathrm{F}}^{\gamma} \wedge m \geq n\right\}$$

from which it is easy to derive the wanted triple using F-DUPLICABLE to create another copy of $\boxed{\mathrm{F}}^{\gamma}$. One of the copies is used to reestablish the invariant $I(\gamma)$, and the other remains in the postcondition.

To conclude the proof of this example we now need to show (18)

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\boxed{\bar{\mathrm{F}}}^{\gamma}\right\} !\ell \{v.v \geq n+1\}$$

Using the invariant opening rule HT-INV-OPEN together with structural rules we need to prove

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\boxed{\bar{\mathrm{F}}}^{\gamma} * \triangleright(\ell \hookrightarrow m * \boxed{\bar{\mathrm{S}}}^{\gamma} \wedge m \geq n)\right\} !\ell \{v.v \geq (n+1) \wedge \triangleright I(\gamma)\}$$
$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\boxed{\bar{\mathrm{F}}}^{\gamma} * \triangleright(\ell \hookrightarrow m * \boxed{\bar{\mathrm{F}}}^{\gamma} \wedge m \geq (n+1))\right\} !\ell \{v.v \geq (n+1) \wedge \triangleright I(\gamma)\}$$

We again prove the first one and leave the second one as an exercise.

Using HT-FRAME-ATOMIC and HT-LOAD we get

$$\boxed{I(\gamma)}^{\iota} \vdash \left\{\boxed{\bar{\mathrm{F}}}^{\gamma} * \triangleright(\ell \hookrightarrow m * \boxed{\bar{\mathrm{S}}}^{\gamma} \wedge m \geq n)\right\} !\ell \left\{v.v = m \wedge \ell \hookrightarrow m * \boxed{\bar{\mathrm{F}}}^{\gamma} * \boxed{\bar{\mathrm{S}}}^{\gamma} \wedge m \geq n\right\}$$

Now by S-F-INCOMPATIBLE, the precondition is equivalent to *later*False, that is, this case is *impossible*, and thus by HT-LATER-FALSE we get the desired triple.

To recap, the high-level idea of the proof is that the invariant can be in two "states". It starts off in the state where we know that the value at ℓ is at least n and then, when incrementing, we transition to a new state, but we also get out a new token, *i.e.*, we get $\boxed{\bar{\mathrm{F}}}^{\gamma}$ in the postcondition. This proposition is then used to decide in which case we are when opening the invariant again. ■

7.4 Ghost state

The ghost state used in the previous section was rather *ad hoc*. If we had to extend the logic with new primitive propositions for each new example, we would need to establish consistency for each such extension. That is not tenable. Thus in this section we develop a very general notion of resources. It is not the most general notion of resources supported by Iris, but the final generalisation is quite technical and postponed until later sections. Consistency of Iris with respect to this notion of resources will be proved in later sections.

To define the notion of resources we need to recall some concepts and facts.

Definition 7.9. A *commutative semigroup* is a set $\mathcal{M}$ together with a function $(\cdot) : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, called the *operation* such that the operation is associative and commutative.

A commutative semigroup is called a *commutative monoid* if there exists an element ε (called the unit) which is the neutral element for the operation $(\cdot)$: for all $m \in \mathcal{M}$, the property $m \cdot \varepsilon = \varepsilon \cdot m = m$ holds.

The set $\mathcal{M}$ is called the *carrier* of the semigroup (resp. monoid). ■

Every semigroup can be made a preorder by defining the *extension order* $a \preccurlyeq b$ as

$$a \preccurlyeq b \iff \exists c, b = a \cdot c.$$

In words, $a \preccurlyeq b$ if *a is a part of b*.

Exercise 7.10. Show that the relation $\preccurlyeq$ is transitive for any semigroup $\mathcal{M}$. Show that it is reflexive if and only if for every element a there exists an element $b \in \mathcal{M}$ such that $a \cdot b = a$. Conclude that if $\mathcal{M}$ is a commutative monoid then $\preccurlyeq$ is reflexive. ◊

Certain kinds of commutative semigroups and monoids serve as good abstract models of resources. Resources can be composed using the operation. Commutativity and associativity

express that the order in which resources are composed does not matter. The unit of the monoid represents the empty resource, which exists in many instances.

Finally, we also need the ability to express that certain resources cannot be combined together. This can be achieved in many ways. The way we choose to do it is to have a subset $\mathcal{V}$ of so-called *valid elements*. Thus, for now, our notion of resources is the following.

Definition 7.11 (Resource algebra)**.** A *resource algebra* is a commutative semigroup $\mathcal{M}$ together with a subset $\mathcal{V} \subseteq \mathcal{M}$ of elements called *valid*, and a *partial* function $|\cdot| : \mathcal{M} \to \mathcal{M}$, called the *core*.

The set of valid elements is required to have the closure property

$$a \cdot b \in \mathcal{V} \Rightarrow a \in \mathcal{V}.$$

The core is required to have the following properties.

$$|a| \text{ defined} \Rightarrow |a| \cdot a = a$$
$$|a| \text{ defined} \Rightarrow ||a|| = |a|$$
$$a \preccurlyeq b \land |a| \text{ defined} \Rightarrow |b| \text{ defined } \land |a| \preccurlyeq |b|.$$

A resource algebra is *unital* if $\mathcal{M}$ is a commutative monoid with unit ε and the following properties hold.

$$\varepsilon \in \mathcal{V} \qquad\qquad |\varepsilon| = \varepsilon.$$

In particular $|\varepsilon|$ is defined. ■

The core of the resource algebra is meant to be a function, which for each element captures the "duplicable part" of an element. Sometimes such a duplicable part does not exist, hence we allow the core to be a partial function; we will see some such examples below.

Exercise 7.12. Show that for any resource algebra $\mathcal{M}$, and any element $a \in \mathcal{M}$, the core of a, if defined, is duplicable, *i.e.*, for any a, show

$$|a| \text{ defined} \Rightarrow |a| \cdot |a| = |a|.$$

◊

Exercise 7.13. Show that in a unital resource algebra the core is always defined. Hint: $\varepsilon \preccurlyeq a$ for any element a. ◊

Example 7.14. A canonical example of a unital resource algebra is the one of heaps. More precisely, the carrier of the resource algebra is the set of heaps plus an additional element, call it $\bot$, which is used to define composition of incompatible heaps. Composition of heaps is disjoint union, and if the heaps are not disjoint, then their composition is defined to be $\bot$. Composing $\bot$ with any other element yields $\bot$. The core is the constant function, mapping every element to the empty heap, which is the unit of the resource algebra. Every heap is valid, the only non-valid element being $\bot$. ■

Example 7.15. We now present an example, which we will use later, and where the core is non-trivial. (It is closely related to the *agreement construction*, which we will also use later on.) Given a set X, the carrier of the resource algebra is the set $X \cup \{\bot\}$, for some element $\bot$ not in X. The operation $(\cdot)$ is defined by the following rules. The non-trivial compositions are only

$$m \cdot m = m$$

and otherwise $m \cdot n = \bot$. The core can be defined as the identity function, and every element apart from $\bot$ is valid. The definition of the core as the identity function is possible since every element of the resource algebra is duplicable. ■

Example 7.16 (Products of resource algebras)**.** If $\mathcal{M}_1$ and $\mathcal{M}_2$ are resource algebras with cores $|\cdot|_1$ and $|\cdot|_2$ and sets of valid elements $\mathcal{V}_1$ and $\mathcal{V}_2$ then we can form the product resource algebra $\mathcal{M}_\times$ with carrier the product of the carriers $\mathcal{M}_1 \times \mathcal{M}_2$ with operation defined component-wise as

$$(a, b) \cdot (a', b') = (a \cdot a', b \cdot b')$$

and the set of valid elements

$$\mathcal{V}_\times = \{(a, b) \mid a \in \mathcal{V}_1, b \in \mathcal{V}_2\}.$$

The core is similarly defined component-wise as

$$|(a, b)|_\times = \begin{cases} (|a|_1, |b|_2) & \text{if } |a|_1 \text{ and } |b|_2 \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

It is easy to see (exercise!) that if both resource algebras are unital then so is the product resource algebra.

This product example can be extended to a product of arbitrary many resource algebras. ■

Example 7.17 (Finite map resource algebra)**.** Let $(\mathcal{M}, \mathcal{V}, |\cdot|)$ be a resource algebra. We can form a new resource algebra $\mathbb{N} \xrightarrow{\text{fin}} \mathcal{M}$ whose carrier is the set of partial functions $\mathbb{N} \to \mathcal{M}$ with finite domain and the operation is defined as

$$(f \cdot g)(n) = \begin{cases} f(n) \cdot g(n) & \text{if } f(n) \text{ and } g(n) \text{ defined} \\ f(n) & \text{if } f(n) \text{ defined and} g(n) \text{ undefined} \\ g(n) & \text{if } g(n) \text{ defined and} f(n) \text{ undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The set of valid finite maps is

$$\mathcal{V}_{\mathbb{N} \xrightarrow{\text{fin}} \mathcal{M}} = \{f \mid \forall n, f(n) \text{ defined} \Rightarrow f(n) \in \mathcal{V}\}$$

and the core is defined as

$$(|f|_{\mathbb{N} \xrightarrow{\text{fin}} \mathcal{M}})(n) = \begin{cases} |f(n)| & \text{if } f(n) \text{ and } |f(n)| \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that $\mathbb{N} \xrightarrow{\text{fin}} \mathcal{M}$ is always a *unital* resource algebra, its unit being the always undefined finite partial function. ■

Example 7.18 (Exclusive resource algebra)**.** Given a set X the exclusive resource algebra $\text{Ex}(X)$ has as carrier the set X with an additional element $\bot$. The operation is defined such that $x \cdot y = \bot$ for all x and y. The core is the always undefined function, and the valid elements are elements of X, *i.e.*, every element of the resource algebra except the $\bot$.

Perhaps it does not seem that this resource algebra is very interesting. In fact it does appear in verification of certain programs, but it can also be used as a building block of other resource algebras, as shown in the following exercise. ■

Exercise 7.19. Show that when restricted to valid elements, the resource algebra $\mathbb{N} \xrightarrow{\text{fin}} \text{Ex}(Val)$ is the same as the unital resource algebra of heaps described in Example 7.14. More precisely, show that the valid elements of $\mathbb{N} \xrightarrow{\text{fin}} \text{Ex}(Val)$ are precisely the heaps, and composition of these is exactly the same as the composition of heaps, if it is a valid element. ◊

Example 7.20 (Option resource algebra)**.** Given any resource algebra (not necessarily unital) $\mathcal{M}$, we define the unital resource algebra $\mathcal{M}_?$. Its carrier is the set $\mathcal{M}$ together with a new element ?. The operation on elements of $\mathcal{M}$ is inherited, and we additionally set $? \cdot x = x \cdot ? = x$, *i.e.*, ? is the unit. The set of valid elements is that of $\mathcal{M}$ and ?. Finally, the core operation is defined as

$$|?|_{\mathcal{M}_?} = ?$$
$$|x|_{\mathcal{M}_?} = \begin{cases} |x| & \text{if } |x| \text{ defined} \\ ? & \text{otherwise} \end{cases}$$

■

With these concepts, we can extend Iris with a general notion of resources, a single unital resource algebra. Strictly speaking the logic is extended with a family of chosen resource algebras $\mathcal{M}_i$, which we leave open, so that new ones can be added when they are needed in the verification of concrete examples. We add the resource algebras, and its elements, the core function, and the property of the element being valid, as new types and new terms of the logic, together with all the equations for the operations. In addition to this we also add the notion of ghost names. These are used to be able to refer to multiple different instances of the same resource algebra element, analogous to how different locations in a heap are used to contain different values.

Thus we extend the logic with the following constructs

$$\frac{\Gamma \vdash a : \mathcal{M}_i \qquad |a|_i \text{ defined}}{\Gamma \vdash |a|_i : \mathcal{M}_i} \qquad \frac{\Gamma \vdash a : \mathcal{M}_i}{\Gamma \vdash a \in \mathcal{V}_i : \mathsf{Prop}} \qquad \frac{\gamma \in \mathsf{GhostName} \qquad \Gamma \vdash a : \mathcal{M}_i}{\Gamma \vdash \boxed{a : \mathcal{M}_i}^{\gamma} : \mathsf{Prop}}$$

The first two are self-explanatory, they internalise the notions of the resource algebra into the logic. The last rule introduces a new construct, the *ghost ownership assertion* $\boxed{a : \mathcal{M}_i}^{\gamma}$, which we will write as $\boxed{a}^{\gamma}$ when the resource algebra $\mathcal{M}_i$ is clear from the context. This assertion states that we own an instance of a ghost resource a named γ.

The rules of the ghost ownership assertion are as follows.

Own-op
$$\boxed{a : \mathcal{M}_i}^{\gamma} * \boxed{b : \mathcal{M}_i}^{\gamma} \dashv\vdash \boxed{a \cdot b : \mathcal{M}_i}^{\gamma}$$

Own-valid
$$\boxed{a : \mathcal{M}_i}^{\gamma} \vdash a \in \mathcal{V}_i$$

And the final rule, which shows why the core is useful, is related to the always modality with the following law of the logic.

Always-core
$$\frac{\Gamma \vdash a : \mathcal{M}_i \qquad |a|_i \text{ defined}}{\boxed{a : \mathcal{M}_i}^{\gamma} \vdash \Box \boxed{|a|_i : \mathcal{M}_i}^{\gamma}}$$

Example 7.21. If we take the resource algebra M to have the carrier $\{\text{S}, \text{F}, \bot\}$ with multiplication defined as $\text{F} \cdot \text{F} = \text{F}$ and otherwise $x \cdot y = \bot$ then, with the rules presented above, we can recover the rules ftok-duplicable, S-S-incompatible and S-F-incompatible, which were postulated in the previous section. The core of the resource algebra M is always undefined. ■

Example 7.22 (Resource algebra of fractions)**.** An often used resource algebra is the one of fractions $\mathbb{Q}_{01}$. Its carrier is the set of (strictly) positive rational numbers q with addition as the operation. However the valid elements are only those q less than 1, *i.e.*, $\mathcal{V} = \{q \mid 0 < q \leq 1\}$. The core is always undefined. ■

Ghost updates We now consider how to update the ghost resources. This ability will be used to evolve the ghost state along with the execution of the program. When the ghost state changes, it is important that it remains valid – Iris always maintains the invariant that the ghost state obtained by composing the contributions of all threads is well-defined and valid, *i.e.*, the all the contributions of all threads must be compatible. We call state changes that maintain this invariant *frame-preserving updates*.

Definition 7.23 (Frame preserving update)**.** For any resource algebra $\mathcal{M}$ with the set of valid elements $\mathcal{V}$ we define a relation, the *frame preserving update* $a \rightsquigarrow B$, where $a \in \mathcal{M}$ and $B \subseteq \mathcal{M}$. It states that any element compatible with a is compatible with *some* element in B. Precisely,

$$a \rightsquigarrow B \iff \forall x \in \mathcal{M}, a \cdot x \in \mathcal{V} \Rightarrow \exists b \in B, b \cdot x \in \mathcal{V}.$$

If B is the singleton set $\{b\}$, we write $a \rightsquigarrow b$ for $a \rightsquigarrow \{b\}$. ■

To support modification of ghost state in the logic we introduce a new *update modality* $\Rrightarrow P$, with associated frame preserving updates. The typing rules and basic axioms of the update modality are show in Figure 9. The intuition is that $\Rrightarrow P$ holds for a resource r, if from r we can do a frame-preserving update to some r' that satisfies P. Thus the update modality $\Rrightarrow P$ provides a way, inside the logic, to talk about the resources we *could* own after performing an update to what we *do* own. With this intuitive reading of $\Rrightarrow P$, the laws in Figure 9 should make sense. For instance, the UPD-FRAME axiom holds because if r satisfies $P * \Rrightarrow Q$, then r can be split into r_1 and r_2 with r_1 in P and r_2 in $\Rrightarrow Q$, and the latter means that r_2 can be updated in a frame-preserving way to some r_2' in Q, *i.e.*, $r_2 \rightsquigarrow r_2'$. But then also $r = (r_1 \cdot r_2) \rightsquigarrow (r_1 \cdot r_2')$ and hence $r \in \Rrightarrow (P * Q)$.

$$\frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \Rrightarrow P : \mathsf{Prop}} \qquad \text{UPD-MONO}\ \frac{P \vdash Q}{\Rrightarrow P \vdash \Rrightarrow Q} \qquad \text{UPD-INTRO}\ \frac{}{P \vdash \Rrightarrow P} \qquad \text{UPD-IDEMP}\ \frac{}{\Rrightarrow \Rrightarrow P \vdash \Rrightarrow P} \qquad \text{UPD-FRAME}\ \frac{}{P * \Rrightarrow Q \vdash \Rrightarrow (P * Q)}$$

Figure 9: Laws for the update modality

Exercise 7.24. Show the following derived rules.

1.
$$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash \Rrightarrow Q_2}{P_1 * P_2 \vdash \Rrightarrow (Q_1 * Q_2)}$$

2.
$$\text{UPD-SEP}\ \frac{P_1 \vdash \Rrightarrow Q_1 \qquad P_2 \vdash \Rrightarrow Q_2}{P_1 * P_2 \vdash \Rrightarrow (Q_1 * Q_2)}$$

3.

$$\frac{P_2 \vdash \Rightarrow Q \qquad P_1 * Q \vdash \Rightarrow R}{P_1 * P_2 \vdash \Rightarrow R}\ \textsc{upd-bind}$$

◊

Remark 7.25. Note that the rule UPD-BIND is a kind of bind or let rule. Indeed, it may be instructive to compare the rule UPD-BIND with the typing rule for a let construct in an ML-like language

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \tau \vdash e_2 : \sigma}{\Gamma \vdash \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \sigma}$$

The difference is that because of the use of separating conjunction we need to separate the resources needed to prove $\Rightarrow Q$ from those needed to prove the $\Rightarrow R$. Thus we cannot use P_2 anymore when proving $\Rightarrow R$.

The following exercise shows that if a more standard let-like rule is added to the logic then the update modality would become significantly weaker. ■

Exercise 7.26. Show that if the rule

$$\frac{P \vdash \Rightarrow Q \qquad P * Q \vdash \Rightarrow R}{P \vdash \Rightarrow R}$$

is added to the logic then the following is derivable for any R.

$$\frac{P * P \vdash \mathsf{False}}{P \vdash \Rightarrow R}$$

In particular $P \vdash \Rightarrow \mathsf{False}$ for P such that $P * P \vdash \mathsf{False}$. ◊

The update modality allows us to allocate and update ghost resources, as explained by the following rules.

$$\frac{a \in \mathcal{V}}{\mathsf{True} \vdash \Rightarrow \exists \gamma. \boxed{a}^{\gamma}}\ \textsc{Ghost-alloc} \qquad \frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \vdash \Rightarrow \boxed{b}^{\gamma}}\ \textsc{Ghost-update}$$

Finally, we connect the update modality with Hoare triples. The idea is that ghost state is abstract state used to keep track of auxiliary facts during proofs. So we should be able to update the ghost state in pre- and postconditions of Hoare triples.

A uniform way to do this is to generalise the consequence rule HT-CSQ. We first define the *view shift* $P \Rrightarrow Q$ as

$$P \Rrightarrow Q = \Box(P \Rightarrow \Rightarrow Q)$$

The generalized rule of consequence is then

$$\frac{S \vdash P' \Rrightarrow P \qquad S \vdash \{P\}\, e\, \{v.Q\} \qquad S \vdash \forall v.\, Q(v) \Rrightarrow Q'(v)}{S \vdash \{P'\}\, e\, \{v.Q'\}}\ \textsc{Ht-csq}$$

Exercise 7.27. Derive the previous rule of consequence from the one just introduced. ◊

Exercise 7.28. Derive the following.

- $$\frac{a \in \mathcal{V}}{P \vdash \Rrightarrow \left(\left(\exists \gamma. \boxed{\overline{a}}^{\gamma}\right) * P\right)}$$
- $$\frac{a \in \mathcal{V}}{\vdash P \Rrightarrow \left(\exists \gamma. \boxed{\overline{a}}^{\gamma}\right) * P}$$

◊

In particular, we have $\mathsf{True} \Rrightarrow \exists \gamma. \boxed{\overline{a}}^{\gamma}$, for all valid a, and if $a \rightsquigarrow b$ then $\boxed{\overline{a}}^{\gamma} \Rrightarrow \boxed{\overline{b}}^{\gamma}$.

Exercise 7.29. Derive the rest of the rules for start and finish tokens used in the previous section for the resource algebra from Example 7.21. That is, show the rules HT-TOKEN-UPDATE-POST, HT-TOKEN-UPDATE-PRE, and HT-TOKEN-ALLOC. ◊

Exercise 7.30 (Allocating invariants in the post-condition)**.** It will often be the case that we need to allocate an invariant in the post-condition, using the following derivable rule.

$$\frac{\begin{array}{c}\text{HT-INV-ALLOC-POST}\\ \mathcal{E} \text{ infinite} \qquad S \vdash \{P_2\}\, e\, \{v.Q\}_{\mathcal{E}}\end{array}}{S \vdash \{(\triangleright P_1) * P_2\}\, e\, \left\{v.Q \wedge \exists \iota \in \mathcal{E}. \boxed{P_1}^{\iota}\right\}_{\mathcal{E}}}$$

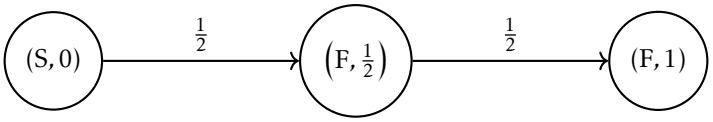
Derive the rule using HT-INV-ALLOC, the fact that invariants are persistent and the generalised rule of consequence introduced above. ◊

Example 7.31. In this example we show how to use slightly more complex reasoning using resource algebras to show the specification (14) (page 46) from the parallel increment example. We are going to use two resource algebras. The two resource algebras we are going to use are the one of fractions detailed in Example 7.22, together with the resource algebra encoding the transition system with states S and F defined in Example 7.21, and used in the previous section.

The proof proceeds similarly to the proof in Example 7.8, but with a different invariant. The invariant we are going to use is

$$\begin{aligned} I(\gamma_1, \gamma_2, n) = \ell \hookrightarrow n * \boxed{\overline{\mathrm{S}}}^{\gamma_1} \vee \\ \ell \hookrightarrow (n+1) * \boxed{\overline{\mathrm{F}}}^{\gamma_1} * \boxed{\overline{\tfrac{1}{2}}}^{\gamma_2} \vee \\ \ell \hookrightarrow (n+2) * \boxed{\overline{\mathrm{F}}}^{\gamma_1} * \boxed{\overline{1}}^{\gamma_2} \end{aligned}$$

That is, we are in essence encoding a three state transition system. The tokens F and S are used to distinguish the initial state from the rest of the states, and the fractions $\frac{1}{2}$ and 1 can be thought of as the price needed to get from the initial state to the current state. We can depict this in the following way



The resources on the transitions can also be viewed as coming from the environment. To make a transition from one state to another the environment will have to give up ownership of $\boxed{\frac{1}{2}}^{\gamma_2}$ and transfer it to the invariant.

With this invariant let us proceed to the proof of the specification

$$\{\ell \hookrightarrow n\}\,(e \,\|\, e);!\ell\,\{v.v = n+1 \vee v = n+2\}.$$

We start off by allocating two pieces of ghost state. We allocate $\boxed{\mathrm{S}}^{\gamma_1}$ and $\boxed{1}^{\gamma_2}$ for some γ_1 and γ_2 using the rule Ghost-alloc and the generalized rule of consequence. Hence we have to show

$$\left\{\boxed{\mathrm{S}}^{\gamma_1} * \boxed{1}^{\gamma_2} * \ell \hookrightarrow n\right\}(e \,\|\, e);!\ell\,\{v.v = n+1 \vee v = n+2\}.$$

Using the invariant allocation rule we allocate the invariant by transferring

$$\boxed{\mathrm{S}}^{\gamma_1} * \ell \hookrightarrow n$$

into the invariant, which then means we have to show.

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\boxed{1}^{\gamma_2}\right\}(e \,\|\, e);!\ell\,\{v.v = n+1 \vee v = n+2\}$$

for some ι. Using the rule Ht-seq we now verify the two parts, showing the following two triples

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\boxed{1}^{\gamma_2}\right\}(e \,\|\, e)\left\{\_.\boxed{\mathrm{F}}^{\gamma_1}\right\}$$
$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\boxed{\mathrm{F}}^{\gamma_1}\right\}(e \,\|\, e);!\ell\,\{v.v = n+1 \vee v = n+2\}.$$

The proof of the second triple is completely analogous to the one in Example 7.8, so we omit it here.

To show the first triple we first use the rule Own-op to get

$$\boxed{1}^{\gamma_2} \iff \boxed{\tfrac{1}{2}}^{\gamma_2} * \boxed{\tfrac{1}{2}}^{\gamma_2}$$

and

$$\boxed{\mathrm{F}}^{\gamma_1} \iff \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\mathrm{F}}^{\gamma_1}$$

and hence the first triple is equivalent to

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\boxed{\tfrac{1}{2}}^{\gamma_2} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\}(e \,\|\, e)\left\{\_.\boxed{\mathrm{F}}^{\gamma_1} * \boxed{\mathrm{F}}^{\gamma_1}\right\}$$

which means we can use the parallel composition rule Ht-par, and we have to show

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\boxed{\tfrac{1}{2}}^{\gamma_2}\right\} e \left\{\_.\boxed{\mathrm{F}}^{\gamma_1}\right\}.$$

Recall that e is the program $\ell \leftarrow !\ell + 1$. Using the Ht-bind rule we show the following two entailments.

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\boxed{\tfrac{1}{2}}^{\gamma_2}\right\} !\ell \left\{v.(v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \tag{19}$$
$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \forall v.\left\{(v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \ell \leftarrow v+1 \left\{\_.\boxed{\mathrm{F}}^{\gamma_1}\right\} \tag{20}$$

To show (19) we open the invariant $I(\gamma_1, \gamma_2, n)$. Thus we get

$$\triangleright\left(\ell \hookrightarrow n * \boxed{\bar{\mathrm{S}}}^{\gamma_1} \vee \ell \hookrightarrow (n+1) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2} \vee \ell \hookrightarrow (n+2) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{1}^{\gamma_2}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2}$$

in the precondition which using the distributivity laws of the logic, together with Own-op simplifies to

$$\begin{aligned}
&\triangleright\left(\ell \hookrightarrow n * \boxed{\bar{\mathrm{S}}}^{\gamma_1}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2} \vee \\
&\triangleright\left(\ell \hookrightarrow (n+1) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2} \vee \\
&\triangleright\left(\ell \hookrightarrow (n+2) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{\tfrac{3}{2}}^{\gamma_2}\right)
\end{aligned}$$

The last disjunct is equivalent to $\triangleright\mathsf{False}$ by Own-valid and the fact that the only valid fractions are those which are not greater than 1. Using the disjunction rule Ht-disj we have to show further three triples, all with postcondition

$$(v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2} * \triangleright I(\gamma_1, \gamma_2, n)$$

and with three preconditions corresponding to the three disjuncts above. The last is the easiest one and follows directly from rules derived in Exercise 7.3. The first two we leave as exercises, since they are direct applications of rules we have seen many times.

Exercise 7.32. Show the following two triples.

$$\left\{\triangleright\left(\ell \hookrightarrow n * \boxed{\bar{\mathrm{S}}}^{\gamma_1}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\}\, !\,\ell\, \left\{v.\, (v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2} * \triangleright I(\gamma_1, \gamma_2, n)\right\}$$

$$\left\{\triangleright\left(\ell \hookrightarrow (n+1) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\}\, !\,\ell\, \left\{v.\, (v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2} * \triangleright I(\gamma_1, \gamma_2, n)\right\}$$

$\diamond$

Let us now turn to showing the specification (20). Using the forall introduction rule, distributivity of $*$ over $\vee$, and Ht-disj this means showing two specifications.

$$\boxed{I(\gamma_1, \gamma_2, n)}^{\iota} \vdash \left\{v = n * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\}\, \ell \leftarrow v + 1\, \left\{\_.\, \boxed{\bar{\mathrm{F}}}^{\gamma_1}\right\} \tag{21}$$

$$\boxed{I(\gamma_1, \gamma_2, n)}^{\iota} \vdash \left\{v = (n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\}\, \ell \leftarrow v + 1\, \left\{\_.\, \boxed{\bar{\mathrm{F}}}^{\gamma_1}\right\} \tag{22}$$

Let us show (21) first. Using Ht-always and ordinary equational reasoning the triple is equivalent to

$$\boxed{I(\gamma_1, \gamma_2, n)}^{\iota} \vdash \left\{\boxed{\tfrac{1}{2}}^{\gamma_2}\right\}\, \ell \leftarrow n + 1\, \left\{\_.\, \boxed{\bar{\mathrm{F}}}^{\gamma_1}\right\}$$

To be completely precise we first need to use the bind rule to compute $n+1$ to a value, but this is completely straighforward, so let us just assume we have done it. We then open the invariant and after simplifying we have

$$\triangleright\left(\ell \hookrightarrow n * \boxed{\bar{\mathrm{S}}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \vee \triangleright\left(\ell \hookrightarrow (n+1) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{1}^{\gamma_2}\right) \vee \triangleright\left(\ell \hookrightarrow (n+2) * \boxed{\bar{\mathrm{F}}}^{\gamma_1} * \boxed{\tfrac{3}{2}}^{\gamma_2}\right)$$

in the precondition. As before the last disjunct is equivalent to ▷False so we may deal with it exactly as before using results from Exercise 7.3. Using HT-DISJ we thus need to prove the following two specifications.

$$\left\{\triangleright\left(\ell \hookrightarrow n * \boxed{\mathrm{S}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right)\right\} \ell \leftarrow n+1 \left\{\_. \boxed{\mathrm{F}}^{\gamma_1} * \triangleright I(\gamma_1,\gamma_2,n)\right\} \tag{23}$$

$$\left\{\triangleright\left(\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{1}^{\gamma_2}\right)\right\} \ell \leftarrow n+1 \left\{\_. \boxed{\mathrm{F}}^{\gamma_1} * \triangleright I(\gamma_1,\gamma_2,n)\right\} \tag{24}$$

To show the first specification we start by using HT-FRAME-ATOMIC to get

$$\left\{\triangleright\left(\ell \hookrightarrow n * \boxed{\mathrm{S}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right)\right\} \ell \leftarrow n+1 \left\{\ell \hookrightarrow n+1 * \boxed{\mathrm{S}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \tag{25}$$

We then use the fact that $\mathrm{S} \rightsquigarrow \mathrm{F}$ together with the rule GHOST-UPDATE, and UPD-FRAME to get

$$\left(\ell \hookrightarrow n+1 * \boxed{\mathrm{S}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \Rrightarrow \left(\ell \hookrightarrow n+1 * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right)$$

Moreover $\mathrm{F} = \mathrm{F} \cdot \mathrm{F}$ and thus OWN-OP gives us $\boxed{\mathrm{F}}^{\gamma_1} \Rrightarrow \left(\boxed{\mathrm{F}}^{\gamma_1} * \boxed{\mathrm{F}}^{\gamma_1}\right)$ and thus together we have

$$\left(\ell \hookrightarrow n+1 * \boxed{\mathrm{S}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \Rrightarrow \left(\ell \hookrightarrow n+1 * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right).$$

Clearly $\ell \hookrightarrow n+1 * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2} \Rrightarrow \triangleright I(\gamma_1,\gamma_2,n)$ and thus we have

$$\left(\ell \hookrightarrow n+1 * \boxed{\mathrm{S}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \Rrightarrow \left(\boxed{\mathrm{F}}^{\gamma_1} * \triangleright I(\gamma_1,\gamma_2,n)\right)$$

and thus finally, by the (generalized) rule of consequence, we get (23) from (25), as needed.

Exercise 7.33. Following similar reasoning illustrated above show specifications (24) and (22) ◊

The proof is somewhat complex, and perhaps the key new point, compared to previous examples, is lost to the reader. The key new way of reasoning in this example is the use of the fraction $\frac{1}{2}$ and how we transferred it from the ownership of the thread to the ownership of the invariant. Technically this can be seen from the fact that if when reading $!\ell$ the invariant was in state $\ell \hookrightarrow n * \boxed{\mathrm{S}}^{\gamma_1}$ then after writing the invariant was in state $\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}$. Analogously, if the invariant was in state $\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}$ when reading then after writing it was in state $\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{1}^{\gamma_2}$. Finally, we have used the fact that the thread owned $\boxed{\tfrac{1}{2}}^{\gamma_2}$ to discount the possibility that we have read the value $n+2$, *i.e.*, that the invariant was in state $\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{1}^{\gamma_2}$. ■

Exercise 7.34. For the same program e as in the preceding example, define an invariant which would allow you to prove the following specification

$$\{\ell \hookrightarrow n\}\ ((e \parallel e) \parallel e); !\ell \{v.v = n+1 \vee v = n+2 \vee v = n+3\}.$$

◊

7.5 Compare and set primitive

The compare and set primitive $\mathsf{cas}(\ell, v_1, v_2)$ is an atomic operation which in one step compares the value stored at location ℓ with v_1. If they agree it stores v_2 to ℓ. Its operational semantics is defined in Section 2. Note that $\mathsf{cas}(\ell, v_1, v_2)$ is *not* equivalent to $\mathsf{if}\,!\ell = v_1\,\mathsf{then}\,\ell \leftarrow v_2$, since the latter expression is not atomic and the value at ℓ can be arbitrary at the time when $\ell \leftarrow v_2$ is executed.

The cas primitive is the basic primitive used to build other synchronisation primitives, such as locks, which we will see in Section 7.6.

The specification of cas is as follows.

HT-CAS

$$\frac{}{\{\triangleright \ell \hookrightarrow v\}\,\mathsf{cas}(\ell, v_1, v_2)\,\{u.(u = \mathsf{true} * v = v_1 * \ell \hookrightarrow v_2) \vee (u = \mathsf{false} * v \neq v_1 * \ell \hookrightarrow v)\}}$$

Often the following derived rules are easier to use.

HT-CAS-SUCC

$$\frac{}{\{\triangleright \ell \hookrightarrow v_1\}\,\mathsf{cas}(\ell, v_1, v_2)\,\{u.u = \mathsf{true} * \ell \hookrightarrow v_2\}}$$

HT-CAS-FAIL

$$\frac{}{\{\triangleright \ell \hookrightarrow v * \triangleright(v \neq v_1)\}\,\mathsf{cas}(\ell, v_1, v_2)\,\{u.u = \mathsf{false} * \ell \hookrightarrow v\}}$$

Exercise 7.35. Derive the rules HT-CAS-SUCC and HT-CAS-FAIL from HT-CAS. ◊

7.6 Examples

Example 7.36 (Spin lock)**.** For our first example of a concurrent program with shared state we will show a specification for a spin lock module. The module consists of three operations, isLock, acquire and release, with the following implementations:

$$\begin{aligned}
&\mathsf{let}\,\mathrm{newLock}() = \mathsf{ref}(\mathsf{false})\\
&\mathsf{let}\,\mathrm{acquire}\,l = \mathsf{if}\,\mathsf{cas}(l, \mathsf{false}, \mathsf{true})\,\mathsf{then}\,()\,\mathsf{else}\,\mathrm{acquire}\,l\\
&\mathsf{let}\,\mathrm{release}\,l = l \leftarrow \mathsf{false}
\end{aligned}$$

Concretely, the lock is a boolean flag, which must be set atomically to indicate that a thread is entering a critical region. We will give an abstract specification, which does not expose the concrete implementation of the lock. Therefore, we specify the operations on the lock using an abstract, *i.e.*, existentially quantified, isLock predicate. The specification we desire for the module as a whole is:

$$\begin{aligned}
&\exists\,\mathrm{isLock} : \mathit{Val} \to \mathsf{Prop} \to \mathsf{GhostName} \to \mathsf{Prop}.\\
&\exists\,\mathrm{locked} : \mathsf{GhostName} \to \mathsf{Prop}.\\
&\quad \forall P, v, \gamma.\ \mathrm{isLock}(v, P, \gamma) \Rightarrow \Box\,\mathrm{isLock}(v, P, \gamma)\\
&\wedge\ \forall \gamma.\ \mathrm{locked}(\gamma) * \mathrm{locked}(\gamma) \Rightarrow \mathsf{False}\\
&\wedge\ \forall P.\ \{P\}\,\mathrm{newLock}()\,\{v.\exists \gamma.\ \mathrm{isLock}(v, P, \gamma)\}\\
&\wedge\ \forall P, v, \gamma.\ \{\mathrm{isLock}(v, P, \gamma)\}\,\mathrm{acquire}\,v\,\{v.P * \mathrm{locked}(\gamma)\}\\
&\wedge\ \forall P, v, \gamma.\ \{\mathrm{isLock}(v, P, \gamma) * P * \mathrm{locked}(\gamma)\}\,\mathrm{release}\,v\,\{_.\mathsf{True}\}
\end{aligned}$$

The specification expresses that the isLock predicate is persistent, hence duplicable, which means that it can be shared among several threads. When a new lock is created using the newLock method, the client obtains an isLock predicate, and the idea is then that since it is duplicable, it can be shared among two (or more) threads which will use the lock to coordinate access to memory shared among the threads. The newLock, acquire, and release methods are all parameterized by a predicate P which describes the resources the lock protects. The postcondition of acquire expresses that once a thread acquires the lock, it gets access to the resources protected by the lock (the P in the postcondition). Moreover, it gets a locked(γ) predicate (think of it as a token), which indicates that it is the current owner of the lock – to call release one needs to have the locked(γ) token. To calls release, a thread needs to have the resources described by P, which are then, intuitively, transferred over to the lock module – the postcondition of release does not include P. Finally, the locked(γ) token is not duplicable, because if it was, it would defeat its purpose of ensuring that only the thread owning the lock would be able to call release. We will discuss an example of a client of the lock module below.

We now proceed to prove that the spin lock implementation meets the above lock module specification.

We need ghost state to record whether the lock is in a locked or an unlocked state. The resource algebra we use is $\{\varepsilon, \bot, \mathrm{K}\}$, with the operation defined as $\varepsilon \cdot x = x \cdot \varepsilon = x$ and otherwise $x \cdot y = \bot$.

To define the isLock predicate, we will make use of an invariant – that will allow us to show that the isLock predicate is persistent, as required by the specification above.

The invariant we use is:

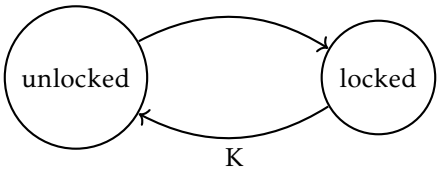
$$I(\ell, P, \gamma) = \ell \hookrightarrow \mathsf{false} * \boxed{\mathrm{K}}^{\gamma} * P \vee \ell \hookrightarrow \mathsf{true}.$$

With this we define the isLock and locked predicates as follows.

$$\begin{aligned} \mathrm{isLock}(v, P, \gamma) &= \exists \ell \in \mathit{Loc}, \iota \in \mathsf{InvName}.\, v = \ell \wedge \boxed{I(\ell, P, \gamma)}^{\iota} \\ \mathrm{locked}(\gamma) &= \boxed{\mathrm{K}}^{\gamma} \end{aligned}$$

The idea of the invariant is as follows. If the location ℓ contains false, then the lock is unlocked. In this case it "owns" the resources P, together with the token K. The K token can be thought of as the "key", which is needed to release, or unlock, the lock. In the post-condition of acquire we obtain locked(γ), and together with the fact that locked(γ) is not duplicable we can ensure that only the thread that acquired the lock has control over releasing it and, moreover, that the lock can only be released once.

We can imagine the resource algebra and the invariant as encoding the following two state labelled transition system.



The label on the transition means that the transition is only valid when we have the token K, *i.e.*, we can only unlock a lock if we have the key.

There are now five proof obligations, one for each of the conjuncts in the specification, and we treat each in turn.

The first says that $\mathsf{isLock}(v, P, \gamma)$ is persistent, which it is because invariants and equality are persistent, and conjunction and existential quantification preserves persistency.

The second says that $\mathsf{locked}(\gamma)$ *is not* duplicable. This follows as $\mathrm{K} \cdot \mathrm{K} = \bot$ by definition of the resource algebra: $\boxed{\mathrm{K}}^{\gamma} * \boxed{\mathrm{K}}^{\gamma} \vdash \boxed{\mathrm{K} \cdot \mathrm{K}}^{\gamma}$ by Own-op which yields False by Own-valid. By transitivity of $\vdash$ we are done.

The third is the specification of allocating a new lock, and hence needs the allocation of a lock invariant. We proceed to show the following triple:

$$\{P\}\, \mathsf{newLock}()\, \{v. \exists \gamma.\, \mathsf{isLock}(v, P, \gamma)\}$$

By Ht-beta, it suffices to show

$$\{P\}\, \mathsf{ref}(\mathsf{false})\, \{v. \exists \gamma.\, \mathsf{isLock}(v, P, \gamma)\}$$

We allocate new ghost state using Ghost-alloc, as in Exercise 7.28, use the rule of consequence and then use Ht-exist. We are left with proving

$$\{\mathsf{locked}(\gamma) * P\}\, \mathsf{ref}(\mathsf{false})\, \{v. \exists \gamma.\, \mathsf{isLock}(v, P, \gamma)\}$$

for some γ.

Exercise 7.37. Prove this. Hint: Use the derived invariant allocation rule Ht-inv-alloc-post. ◊

The fourth is the specification of the acquire operation. It is a recursive definition, so we proceed with the derived rule for recursive functions from Exercise 6.4. That is, assuming

$$\forall v, P, \gamma.\, \{\triangleright\, \mathsf{isLock}(v, P, \gamma)\}\, \mathsf{acquire}\, v\, \{v. P * \mathsf{locked}(\gamma)\} \tag{26}$$

we show the following triple

$$\{\mathsf{isLock}(v, P, \gamma)\}\, \mathsf{if}\, \mathsf{cas}(v, \mathsf{false}, \mathsf{true})\, \mathsf{then}\, ()\, \mathsf{else}\, \mathsf{acquire}(v)\, \{v. P * \mathsf{locked}(\gamma)\}.$$

The isLock predicate gives us that v is a location ℓ governed by an invariant, which we can move into the context as follows:

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \{\mathsf{True}\}\, \mathsf{if}\, \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})\, \mathsf{then}\, ()\, \mathsf{else}\, \mathsf{acquire}(\ell)\, \{v. P * \mathsf{locked}(\gamma)\}$$

We next evaluate the cas with the Ht-bind rule. As our intermediate step we proceed to show the following triple:

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \{\mathsf{True}\}\, \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true})\, \{u. (u = \mathsf{true} * P * \mathsf{locked}(\gamma)) \vee (u = \mathsf{false})\}.$$

As cas is atomic, we open the invariant to get at ℓ, using Ht-inv-open, and it suffices to show that

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \begin{array}{l} \{\triangleright\, I(\ell, P, \gamma)\} \\ \quad \mathsf{cas}(\ell, \mathsf{false}, \mathsf{true}) \\ \{u. ((u = \mathsf{true} * P * \mathsf{locked}(\gamma)) \vee (u = \mathsf{false})) * I(\ell, P, \gamma)\} \end{array}.$$

We proceed by cases on the invariant (using the rule Ht-disj). In the first case we need to show

$$\{\triangleright(\ell \hookrightarrow \mathsf{false} * \mathrm{locked}\,\gamma * P)\}$$

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \quad \mathsf{cas}(\ell,\mathsf{false},\mathsf{true}) \qquad .$$

$$\{u.(u = \mathsf{true} * P * \mathrm{locked}(\gamma) \vee (u = \mathsf{false})) * I(\ell,P,\gamma)\}$$

By Ht-csq it suffices to establish either choice of the disjunctions in the postcondition (there is one in the left of the separating conjuction, and one to the right, hidden in $I(\ell,P,\gamma)$). We choose $u = \mathsf{true} * P * \mathrm{locked}(\gamma) * \ell \hookrightarrow \mathsf{true}$ and by Ht-frame and Ht-CAS-succ we are done.

In the second case, we show

$$\{\triangleright(\ell \hookrightarrow \mathsf{true})\}$$

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \quad \mathsf{cas}(\ell,\mathsf{false},\mathsf{true}) \qquad .$$

$$\{u.((u = \mathsf{true} * P * \mathrm{locked}(\gamma) \vee (u = \mathsf{false})) * I(\ell,P,\gamma)\}$$

Again we strengthen the post-condition, this time to $u = \mathsf{false} * \ell \hookrightarrow \mathsf{true}$, and we are directly done by Ht-CAS-fail.

We are now ready to proceed with our use of Ht-bind, the evaluation of the if, and the following obligation remains:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{u = \mathsf{true} * P * \mathrm{locked}(\gamma) \vee u = \mathsf{false}\}\,\mathsf{if}\,u\,\mathsf{then}\,()\,\mathsf{else}\,\mathrm{acquire}\,\ell\,\{\_.P * \mathrm{locked}(\gamma)\}$$

We consider the two cases in the precondition, using Ht-disj. We use Ht-If-True and Ht-If-False in the first and second case respectively, which leaves the following two obligations:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{P * \mathrm{locked}(\gamma)\}\,()\,\{\_.P * \mathrm{locked}(\gamma)\}$$
$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\mathsf{True}\}\,\mathrm{acquire}\,\ell\,\{\_.P * \mathrm{locked}(\gamma)\}$$

The first follows by the rule for the unit expressions, the second by our induction hypothesis (26). This concludes the proof that acquire satisfies its specification.

The fifth and final is the specification of the release operation. We proceed to show the following triple:

$$\{\mathrm{isLock}(v,P,\gamma) * P * \mathrm{locked}(\gamma)\}\,\mathrm{release}\,v\,\{\_.\mathsf{True}\}$$

By definition, $\mathrm{isLock}(v,P,\gamma)$ tells us there is a location governed by an invariant, and we can substitute this location into the expression under evaluation, and by Ht-beta we can unfold the definition of release:

$$\left\{\boxed{I(\ell,P,\gamma)}^{\iota} * P * \mathrm{locked}(\gamma)\right\}\ell \leftarrow \mathsf{false}\,\{\_.\mathsf{True}\}$$

To perform the assignment we must obtain ℓ as a resource from the invariant, which we do by opening it with the Ht-inv-open rule. As invariants are persistent, we can move it into our assumptions before opening, leaving us with the following triple:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\triangleright I(\ell,P,\gamma) * P * \mathrm{locked}(\gamma)\}\,\ell \leftarrow \mathsf{false}\,\{\_.\triangleright I(\ell,P,\gamma)\}$$

We consider two cases, based on the disjunction in $I(\ell, P, \gamma)$ in the precondition. The first case is

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow \mathsf{false} * \mathrm{locked}(\gamma) * P) * P * \mathrm{locked}(\gamma)\}\, \ell \leftarrow \mathsf{false}\, \{\_.\, \triangleright I(\ell,P,\gamma)\}$$

which is inconsistent as $\mathrm{locked}(\gamma) * \mathrm{locked}(\gamma) \vdash \mathsf{False}$, as argued above. We are done by HT-LATER-FALSE. In the second case we need to prove

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow \mathsf{true}) * P * \mathrm{locked}(\gamma)\}\, \ell \leftarrow \mathsf{false}\, \{\_.\, \triangleright I(\ell,P,\gamma)\}$$

and in the postcondition we chose the first disjunct by HT-CSQ – *i.e.*, we will show the following triple:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow \mathsf{true}) * \triangleright(P * \mathrm{locked}(\gamma))\}\, \ell \leftarrow \mathsf{false}\, \{\_.\, \triangleright(\ell \hookrightarrow \mathsf{false}) * \triangleright(\mathrm{locked}(\gamma) * P)\}$$

which holds by the frame rule and HT-STORE. ■

To show how the lock specification can be used, we use it in an example program: We implement a concurrent bag, using the spin lock to guard access to a shared location containing the data in the bag.

Example 7.38 (Concurrent coarse-grained bag)**.** The implementation is as follows. Recall we use syntactic sugar None for $\mathsf{inj}_1\,()$ and Some x for $\mathsf{inj}_2\, x$. The newBag method allocates a new reference cell which initially contains None, together with a new lock. This lock is used to guard access to the location in insert and remove methods. The location will always contain a list of values. The insert and remove methods insert and remove elements. The remove method returns either None, in the case the bag is empty, or Some v, where v is the head element of the non-empty list.

$$\begin{array}{rl}
\mathsf{let}\,\mathrm{newBag} = \lambda_.\,(\mathsf{ref}(\mathrm{None}), \mathrm{newLock}()) & \\
\mathsf{let}\,\mathrm{insert} = \lambda x.\,\lambda v.\,\mathsf{let}\,\ell = \pi_1\, x\,\mathsf{in} & \\
\qquad\qquad \mathsf{let}\,\mathrm{lock} = \pi_2\, x\,\mathsf{in} & \\
\qquad\qquad \mathrm{acquire}\,\mathrm{lock}; & \\
\qquad\qquad \ell \leftarrow \mathrm{Some}(v, !\,\ell); & \\
\qquad\qquad \mathrm{release}\,\mathrm{lock} & \\
\mathsf{let}\,\mathrm{remove} = \lambda x.\,\mathsf{let}\,\ell = \pi_1\, x\,\mathsf{in} & \\
\qquad\qquad \mathsf{let}\,\mathrm{lock} = \pi_2\, x\,\mathsf{in} & \\
\qquad\qquad \mathrm{acquire}\,\mathrm{lock}; & \\
\qquad\qquad \mathsf{let}\, r = \mathsf{match}\, !\,\ell\, \mathsf{with} & \\
\qquad\qquad\qquad \mathrm{None} \Rightarrow \mathrm{None} & \\
\qquad\qquad\qquad \mid \mathrm{Some}\, p \Rightarrow \ell \leftarrow \pi_2\, p; \mathrm{Some}(\pi_1\, p) & \\
\qquad\qquad\qquad \mathsf{end} & \\
\qquad\qquad \mathsf{in}\ \mathrm{release}\,\mathrm{lock}; r &
\end{array}$$

We would like to have a specification of the bag, which will allow clients to use it in a concurrent setting, where different threads to insert and remove elements from a bag.

A weak, but still useful specification is the following. Given a predicate Φ, the bag contains elements x for which $\Phi(x)$ holds. When inserting an element we give away the resources, and

when removing an element we give back an element plus the knowledge that it satisfies the predicate. The specification is:

$$
\begin{aligned}
&\exists\, \mathsf{isBag} : (\mathit{Val} \to \mathsf{Prop}) \times \mathit{Val} \to \mathsf{Prop}.\\
&\forall(\Phi : \mathit{Val} \to \mathsf{Prop}).\\
&\qquad \forall b.\ \mathsf{isBag}(\Phi, b) \Rrightarrow \Box\mathsf{isBag}(\Phi, b)\\
&\wedge \quad \{\mathsf{True}\}\ \mathsf{newBag}()\,\{b.\ \mathsf{isBag}(\Phi, b)\}\\
&\wedge \quad \forall b u.\ \{\mathsf{isBag}(\Phi, b) * \Phi(u)\}\ \mathsf{insert}\, b\, u\, \{\_.\mathsf{True}\}\\
&\wedge \quad \forall b.\ \{\mathsf{isBag}(\Phi, b)\}\ \mathsf{remove}\, b\, \{v. v = \mathsf{None} \vee \exists x.\ v = \mathsf{Some}\, x \wedge \Phi(x)\}
\end{aligned}
$$

With this specification, the only thing we get to know after calling remove is that the returned element, if we get one out, satisfies the chosen predicate Φ. In fact, giving a stronger specification is hard. The reason is that the isBag predicate is freely duplicable. And we do want the isBag to be duplicable, since this allows us to share it between as many threads as we need, which in turn allows us to specify and prove concurrent programs. The consequence of isBag being duplicable is that concurrently running threads will be able to add and remove elements, so each thread has no guarantee which particular elements it will get back when calling remove.

We now proceed to show that the implementation meets the specification. The isBag predicate is defined as follows:

$$\mathsf{isBag}(\Phi, b) = \exists \ell v \gamma.\ b = (\ell, v) \wedge \mathsf{isLock}(v, \exists xs.\ \ell \hookrightarrow xs * \mathsf{bagList}(\Phi, xs), \gamma)$$

where bagList is defined by guarded recursion as the unique predicate satisfying

$$\mathsf{bagList}(\Phi, xs) = xs = \mathsf{None} \vee \exists x.\ \exists r.\ xs = \mathsf{Some}(x, r) \wedge \Phi(x) * \triangleright(\mathsf{bagList}(\Phi, r)).$$

Let $\Phi : \mathit{Val} \to \mathsf{Prop}$ be arbitrary.

Exercise 7.39. Prove that $\mathsf{isBag}(\Phi, b)$ is persistent for any b. ◊

Exercise 7.40. Prove the newBag specification:

$$\{\mathsf{True}\}\ \mathsf{newBag}()\,\{b.\ \mathsf{isBag}(\Phi, b)\}.$$

◊

Note that since $\mathsf{isBag}(\Phi, b)$ is persistent for any b we can derive, by using the frame rule (exercise!), the following specification

$$\forall b.\ \{\mathsf{isBag}(\Phi, b)\}\ \mathsf{remove}\, b\, \{v.\ (v = \mathsf{None} \vee \exists x.\ v = \mathsf{Some}\, x \wedge \Phi(x)) * \mathsf{isBag}(\Phi, b)\}$$

from the one claimed above, *i.e.*, we do not lose the knowledge that b is a bag.

Let us now prove the specification of the remove method. We are proving

$$\{\mathsf{isBag}(\Phi, b)\}\ \mathsf{remove}\, b\, \{v. v = \mathsf{None} \vee \exists x.\ v = \mathsf{Some}\, x \wedge \Phi(x)\}$$

for some value b.

By definition of $\mathsf{isBag}(\Phi, b)$ and by using Ht-exist, and then Ht-always together with Ht-Eq we have to prove

$$\{\mathsf{isLock}(\mathsf{lock}, \exists xs.\ \ell \hookrightarrow xs * \mathsf{bagList}(\Phi, xs), \gamma)\}\ \mathsf{remove}(\ell, \mathsf{lock})\, \{u. u = \mathsf{None} \vee \exists x.\ u = \mathsf{Some}\, x \wedge \Phi(x)\}$$

for some ℓ, lock and γ. Using HT-BETA and HT-LET-DET we reduce to showing

$$\{\mathsf{isLock}(\mathsf{lock}, \exists xs.\, \ell \hookrightarrow xs * \mathsf{bagList}(\Phi, xs), \gamma)\}\, e\, \{u.u = \mathsf{None} \vee \exists x.\, u = \mathsf{Some}\, x \wedge \Phi(x)\}$$

where e is the program

$$\begin{array}{l}\mathsf{acquire}\,\mathsf{lock};\\ \mathsf{let}\ r = \ \mathsf{match}\,!\ell\,\mathsf{with}\\ \qquad\quad \mathsf{None} \quad \Rightarrow \mathsf{None}\\ \qquad\quad \mid \mathsf{Some}\, p \Rightarrow \ell \leftarrow \pi_2\, p; \mathsf{Some}(\pi_1\, p)\\ \qquad\quad \mathsf{end}\\ \mathsf{in}\ \mathsf{release}\,\mathsf{lock}; r\end{array}$$

Using the sequencing rule HT-SEQ we use the specification of acquire as the first triple, and thus we have to prove

$$\{\mathsf{locked}(\gamma) * \exists xs.\, \ell \hookrightarrow xs * \mathsf{bagList}(\Phi, xs)\}\, e'\, \{u.u = \mathsf{None} \vee \exists x.\, u = \mathsf{Some}\, x \wedge \Phi(x)\}$$

where e' is the part of program e after acquire.

Using the fact that $\exists$ and $\vee$ distribute over $*$ (see Figure 4 on page 13), HT-EXIST and then the definition of $\mathsf{bagList}(\Phi, xs)$ together with HT-DISJ we consider two cases. The first case is

$$\{\mathsf{locked}(\gamma) * \ell \hookrightarrow xs * xs = \mathsf{None}\}\, e'\, \{u.u = \mathsf{None} \vee \exists x.\, u = \mathsf{Some}\, x \wedge \Phi(x)\}$$

Exercise 7.41. Prove the above triple (possibly after looking at the proof of the second case). ◊

In the second case, after using rules in Figure 4 and HT-EXIST, we get the following proof obligation.

$$\{\mathsf{locked}(\gamma) * \ell \hookrightarrow xs * xs = \mathsf{Some}(x, r) * \Phi(x) * \triangleright\, \mathsf{bagList}(\Phi, r)\}\, e'\, \{u.u = \mathsf{None} \vee \exists x.\, u = \mathsf{Some}\, x \wedge \Phi(x)\}$$

which simplifies, by HT-EQ and the rule of consequence, to proving

$$\{\mathsf{locked}(\gamma) * \ell \hookrightarrow \mathsf{Some}(x, r) * \Phi(x) * \triangleright\, \mathsf{bagList}(\Phi, r)\}\, e'\, \{u.\exists x.\, u = \mathsf{Some}\, x \wedge \Phi(x)\}.$$

We use the let rule HT-LET-DET. For the first premise we show

$$\begin{array}{l}\{\mathsf{locked}(\gamma) * \ell \hookrightarrow \mathsf{Some}(x, r) * \Phi(x) * \triangleright\, \mathsf{bagList}(\Phi, r)\}\\ \quad \mathsf{match}\,!\ell\,\mathsf{with}\\ \quad\ \mathsf{None} \quad \Rightarrow \mathsf{None}\\ \quad \mid \mathsf{Some}\, p \Rightarrow \ell \leftarrow \pi_2\, p; \mathsf{Some}(\pi_1\, p)\\ \quad \mathsf{end}\\ \{u.u = \mathsf{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathsf{locked}(\gamma) * \mathsf{bagList}(\Phi, r)\}\end{array}$$

(note the omission of $\triangleright$ on bagList in the postcondition). We start with the bind rule, then the HT-LOAD rule and then HT-MATCH which means we have to show

$$\begin{array}{l}\{\mathsf{locked}(\gamma) * \ell \hookrightarrow \mathsf{Some}(x, r) * \Phi(x) * \triangleright\, \mathsf{bagList}(\Phi, r)\}\\ \quad \ell \leftarrow \pi_2(x, r); \mathsf{Some}(\pi_1(x, r))\\ \{u.u = \mathsf{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathsf{locked}(\gamma) * \mathsf{bagList}(\Phi, r)\}\end{array}$$

which by using HT-BIND and HT-PROJ simplifies to showing

$$\begin{array}{l}\{\mathrm{locked}(\gamma) * \ell \hookrightarrow \mathrm{Some}(x, r) * \Phi(x) * \triangleright\, \mathrm{bagList}(\Phi, r)\} \\ \quad \ell \leftarrow r; \mathrm{Some}(\pi_1(x, r)) \\ \{u.u = \mathrm{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi, r)\}\end{array}$$

Using the sequencing rule we first show

$$\begin{array}{l}\{\mathrm{locked}(\gamma) * \ell \hookrightarrow \mathrm{Some}(x, r) * \Phi(x) * \triangleright\, \mathrm{bagList}(\Phi, r)\} \\ \quad \ell \leftarrow r \\ \{\_.\ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi, r)\}\end{array}$$

by using HT-FRAME-ATOMIC to remove the $\triangleright$ on bagList. The triple, the second premise of the consequence rule,

$$\begin{array}{l}\{\ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi, r)\} \\ \quad \mathrm{Some}(\pi_1(x, r)) \\ \{u.u = \mathrm{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi, r)\}\end{array}$$

is then easy to establish. Exercise!

Finally, we need to establish the second premise of the rule HT-LET-DET, which means showing

$$\begin{array}{l}\{\ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi, r)\} \\ \quad \mathrm{release}\, \mathrm{lock}; \mathrm{Some}\, x \\ \{u.\exists x.\, u = \mathrm{Some}\, x \wedge \Phi(x)\}\end{array}$$

We can use the sequencing rule together with the release specification to give away the resources $\ell \hookrightarrow r$, $\mathrm{locked}(\gamma)$ and $\mathrm{bagList}(\Phi, r)$ back to the lock. We are left with proving

$$\begin{array}{l}\{\Phi(x)\} \\ \quad \mathrm{Some}\, x \\ \{u.\exists x.\, u = \mathrm{Some}\, x \wedge \Phi(x)\}\end{array}$$

which is immediate.

Note that even if we did not call release we could have proved the same triple, by using weakening, *i.e.*, the rule $P * Q \vdash P$. Such an implementation would indeed be safe, but we could never remove more than one element from the bag, since we would not be able to acquire a lock more than once. This is a general observation about an affine program logic such as Iris.

Exercise 7.42. Prove the specification of the insert method:

$$\forall b u.\, \{\mathrm{isBag}(\Phi, b) * \Phi(u)\}\ \mathrm{insert}\, b\, u\ \{\_.\mathsf{True}\}$$

◊

■

7.7 Authoritative resource algebra: counter modules

In this section we will endeavour to specify a counter module that can be used simultaneously by many different threads. We will see that to achieve this we will have to introduce a new kind of resource algebra, the *authoritative resource algebra*.

A first attempt at a specification is as follows. The counter module has three methods, newCounter for creating a fresh counter, incr for increasing the value of the counter, and read for reading the current value of the counter. There is an abstract predicate isCounter(v, n) which should state that v is a counter whose current value is n. This predicate should be persistent, so different threads can access the counter simultaneously. For this reason isCounter(v, n) cannot state that n is *exactly* the value of the counter, but only its lower bound. The reason we can specify the lower bound is that the counter can only increase in value. Hence in particular other threads can only increase the value of the counter, hence they can never invalidate our lower bound.

With this, the specification of the incr method is straightforward: the lower bound of the value of the counter is increased by 1.

$$\forall v.\, \forall n.\, \{\text{isCounter}(v, n)\}\ \text{incr}\, v\, \{u.u = () * \text{isCounter}(v, n+1)\}.$$

Following the discussion above, when reading the value of the counter we do know the exact value, but only its lower bound.

$$\forall v.\, \forall n.\, \{\text{isCounter}(v, n)\}\ \text{read}\, v\, \{u.u \geq n\}$$

The counter implementation we have in mind is the following. The newCounter method creates the counter, which is simply a location containing the counter value.

$$\text{newCounter}() = \ \mathsf{ref}(0)$$

The incr method increases the value of the counter by 1. Since $\ell \leftarrow !\ell + 1$ is not an atomic operation we use a cas loop, as seen in examples before.

$$\begin{aligned}\mathsf{rec}\,\text{incr}(\ell) = {}& \mathsf{let}\, n = !\,\ell\, \mathsf{in}\\ & \mathsf{let}\, m = n + 1\, \mathsf{in}\\ & \mathsf{if}\, \mathsf{cas}(\ell, n, m)\, \mathsf{then}\, ()\, \mathsf{else}\, \text{incr}\, \ell\end{aligned}$$

Finally the read method simply reads the value

$$\text{read}\, \ell = !\,\ell.$$

Now, what should the isCounter predicate be? A first attempt might be simply

$$\text{isCounter}(\ell, n) = \ell \hookrightarrow n.$$

However this clearly cannot work, since such an isCounter predicate is not persistent. A second attempt might be to put the points-to assertion into the invariant as

$$\text{isCounter}(\ell, n) = \exists \iota.\, \boxed{\ell \hookrightarrow n}^{\iota}.$$

This gets us closer, but the problem is that $\ell \hookrightarrow n$ is not an invariant of all the methods, *i.e.*, not all the methods maintain this invariant. In particular the increment method does not. Recall that we can never change the assertion stored in the invariant, *e.g.*, we cannot change $\boxed{\ell \hookrightarrow n}^{\iota}$

into $\boxed{\ell \hookrightarrow (n+1)}^{\iota}$. What is an invariant is, for example, $\exists m.\, \ell \mapsto m$, but now we need to somehow relate m to n which is the parameter of the isCounter predicate. An idea might be to use the invariant

$$\exists m.\, \ell \mapsto m \land m \geq n$$

and thus have

$$\text{isCounter}(\ell, n) = \exists \iota. \boxed{\exists m.\, \ell \mapsto m \land m \geq n}^{\iota}.$$

This is an invariant, however it is not strong enough. We cannot prove the increment method using this invariant since we cannot update $\text{isCounter}(\ell, n)$ to $\text{isCounter}(\ell, n+1)$, because we cannot change the assertion in the invariant. We can conclude that any attempt which mentions n in the invariant directly must fail, so we need some other way to relate the real counter value m and the lower bound n. We will use ghost state. The idea is that in the invariant we will have some ghost state dependent on m, let us call it $\boxed{\bullet m}^{\gamma}$, whereas we will keep some other piece of ghost state in the $\text{isCounter}(\ell, n)$ predicate outside the invariant. Let us call this piece $\boxed{\circ n}^{\gamma}$. Let us see what we need from the resource algebra to be able to verify the counter example by using

$$\text{isCounter}(\ell, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists \iota. \boxed{\exists m.\, \ell \mapsto m * \boxed{\bullet m}^{\gamma}}^{\iota}$$

as the abstract predicate. First, to verify the read method we will open the invariant, and after some simplification we will have $\boxed{\bullet m \cdot \circ n}^{\gamma}$ and the value we are going to return is m. From this we should be able to conclude that $m \geq n$. Using Own-valid we have $\bullet m \cdot \circ n \in \mathcal{V}$, where $\mathcal{V}$ is the set of valid elements of the resource algebra we are trying to define. Hence we need to be able to conclude $m \geq n$ from $\bullet m \cdot \circ n \in \mathcal{V}$. Next, if $\text{isCounter}(\ell, n, \gamma)$ is to be persistent, it must be that $\boxed{\circ n}^{\gamma}$ is persistent, which is only the case (see Always-core) if $|\circ n| = \circ n$, where $|\cdot|$ is the core operation of the resource algebra we are defining. In particular this means that $\circ n$ must be duplicable for any n.

Finally, let us see what we need to verify the incr method. As we have seen many times by now, we have to update the ghost state when the cas operation succeeds. Just before the cas operation succeeds the following resources available

$$\ell \hookrightarrow k * \boxed{\bullet k \cdot \circ n}^{\gamma}$$

and just after we will have

$$\ell \hookrightarrow (k+1) * \boxed{\bullet k \cdot \circ n}^{\gamma}.$$

Using these we need to reestablish the invariant, and get $\boxed{\circ (n+1)}^{\gamma}$ in order to conclude

$$\text{isCounter}(\ell, n+1, \gamma).$$

The only way to do this is to update the ghost state using Ghost-update. Thus it would suffice to have the frame preserving update

$$\bullet k \cdot \circ n \rightsquigarrow \bullet (k+1) \cdot \circ (n+1).$$

To recap, here are the requirements of our resource algebra.

$$|\circ n| = \circ n \tag{27}$$

$$\bullet m \cdot \circ n \in \mathcal{V} \Rightarrow m \geq n \tag{28}$$

$$\bullet m \cdot \circ n \rightsquigarrow \bullet (m+1) \cdot \circ (n+1) \tag{29}$$

We now define a resource algebra which allows us to achieve these properties. Let $\mathcal{M} = \mathbb{N}_{\bot,\top} \times \mathbb{N}$ where $\mathbb{N}_{\bot,\top}$ is the set of natural numbers with two additional elements $\bot$ and $\top$. Define the operation $\cdot$ as

$$(x,n)\cdot(y,m) = \begin{cases} (y, \max(n,m)) & \text{if } x = \bot \\ (x, \max(n,m)) & \text{if } y = \bot \\ (\top, \max(n,m)) & \text{otherwise} \end{cases}$$

It is easy to see (exercise!) that this makes $\mathcal{M}$ into a commutative semigroup. Moreover it has a unit, which is the element $(\bot, 0)$.

For $m, n \in \mathbb{N}$ let us write $\bullet m$ for $(m,0)$ and $\circ n$ for $(\bot, n)$. Using the definition of the operation we clearly see $\bullet m \cdot \circ n = (m,n)$. Thus to get property (28) we should require that if $(m,n) \in \mathcal{V}$ for natural numbers n and m then $m \geq n$. Moreover, the closure condition of the set of valid elements states that subparts of valid elements must also be valid. Thus, since we wish $(n,n) = \circ n \cdot \bullet n \in \mathcal{V}$ we must also have $\circ n = (\bot, n) \in \mathcal{V}$. With this in mind we define the set of valid elements as

$$\mathcal{V} = \{(x,n) \mid x = \bot \vee x \in \mathbb{N} \wedge x \geq n\}.$$

In particular note that elements of the form $(\top, n)$ are *not* valid. With this definition we can see that property (28) holds.

Requirement (27) defines the core on elements $\circ n$. The only way to extend it to the whole $\mathcal{M}$ so that it still satisfies all the axioms of the core is to define

$$|(x,n)| = (\bot, n).$$

It is easy to see that this definition makes $(\mathcal{M}, \mathcal{V}, |\cdot|)$ into a unital resource algebra.

Finally, let us check we have property (29). Recall Definition 7.23 (on page 59) of frame preserving updates. Let $(x,y) \in \mathcal{M}$ be such that $(\bullet m \cdot \circ n)\cdot(x,y)$ is valid. This means in particular that $x = \bot$ and that $m \geq \max(n,y)$. Hence $m+1 \geq \max(n+1, y)$ and thus $(\bullet(m+1)\cdot\circ(n+1))\cdot(x,y)$ is also valid, as needed.

In particular note how it was necessary that $\bullet m \cdot \bullet k = (\top, 0)$ is *not* valid to conclude that x must be $\bot$, which is why we define the operation in this rather peculiar way.

Exercise 7.43. Let isCounter : $Val \to \mathbb{N} \to \mathsf{GhostName} \to \mathsf{Prop}$ be the predicate

$$\mathrm{isCounter}(\ell, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists \iota. \boxed{\exists m.\, \ell \mapsto m * \boxed{\bullet m}^{\gamma}}^{\iota}.$$

Show the following specifications for the methods defined above.

$$\{\mathsf{True}\}\ \mathrm{newCounter}()\ \{u. \exists \gamma.\ \mathrm{isCounter}(u, 0, \gamma)\}$$
$$\forall \gamma.\, \forall v.\, \forall n.\, \{\mathrm{isCounter}(v, n, \gamma)\}\ \mathrm{read}\, v\ \{u. u \geq n\}$$
$$\forall \gamma.\, \forall v.\, \forall n.\, \{\mathrm{isCounter}(v, n, \gamma)\}\ \mathrm{incr}\, v\ \{u. u = () * \mathrm{isCounter}(v, n+1, \gamma)\}$$

♢

Exercise 7.44. Let e be the program

$$\mathsf{let}\, c = \mathrm{newCounter}()\, \mathsf{in}\, (\mathrm{incr}\, c \| \mathrm{incr}\, c); \mathrm{read}\, c.$$

Using the specification of the counter module from the preceding exercise show the following specification for e.

$$\{\mathsf{True}\}\, e\, \{v. v \geq 2\}.$$

♢

A more precise counter specification

The specification of the program e from the above exercise is the strongest possible given the specification of the counter from Exercise 7.43. However operationally we know that the result of that program is exactly the value 2. With the isCounter predicate as above we cannot prove such a precise result simply because isCounter is freely duplicable, and so in each thread we do not know that there are no other thread using the counter, and possibly increasing its value.

In order to give a more precise specification to the counter we must keep track of whether we are the only ones who currently has access to the counter, or if there are possibly other threads using it. This can be achieved by using fractions in the following way. The isCounter will be parametrized by q which indicates the degree of ownership of the counter. If we own the full counter, *i.e.*, $q = 1$ then we know its exact value. If we own only a part of it then we only know its lower bound. The read method thus has two specification.

$$\forall \gamma.\, \forall v.\, \forall n.\, \{\text{isCounter}(v, n, \gamma, 1)\}\ \text{read}\, v\, \{u.u = n\}$$
$$\forall q.\, \forall \gamma.\, \forall v.\, \forall n.\, \{\text{isCounter}(v, n, \gamma, q)\}\ \text{read}\, v\, \{u.u \geq n\}$$

The increment has the same specification, apart from it being parametrized by q, and the newCounter method creates a counter which is owned in full by the thread that created it.

The next question is how do we share the counter. As explained above, the isCounter predicate cannot be persistent. Instead, when splitting the counter we must record that somebody else can also use it. This is achieved by requiring the isCounter predicate to have the following property.

$$\text{isCounter}(v, n, \gamma, p) * \text{isCounter}(v, m, \gamma, q) \dashv\vdash \text{isCounter}(v, n + m, \gamma, p + q).$$

In light of this rule there is another way to read the assertion $\text{isCounter}(v, n, \gamma, p)$. We can read it as that the contribution of this thread to the total value of the counter is exactly n. If p is 1 then this is the only thread, and so the value of the counter is exactly n.

To define the desired isCounter predicate and to prove the desired specification we will need a resource algebra similar to the one used for the first counter specification, but more involved. To achieve it we need to generalize the resource algebra we have defined above to the construction called *authoritative resource algebra*.

Example 7.45 (Authoritative resource algebra)**.** Given a *unital* resource algebra $\mathcal{M}$ with unit ε, set of valid elements $\mathcal{V}$ and core $|\cdot|$, let $\textsc{Auth}(\mathcal{M})$ be the resource algebra whose carrier is the set $\mathcal{M}_{\bot,\top} \times \mathcal{M}$ (recall that $\mathcal{M}_{\bot,\top}$ is the set $\mathcal{M}$ together with two new elements $\bot$ and $\top$) and whose operation is defined as

$$(x, a) \cdot (y, b) = \begin{cases} (y, a \cdot b) & \text{if } x = \bot \\ (x, a \cdot b) & \text{if } y = \bot \\ (\top, a \cdot b) & \text{otherwise} \end{cases}$$

The core function is defined as (recall that the core is total in a unital resource algebra; see Exercise 7.13)

$$|(x, a)|_{\textsc{Auth}(\mathcal{M})} = (\bot, |a|)$$

and the set of valid elements is

$$\mathcal{V}_{\textsc{Auth}(\mathcal{M})} = \left\{ (x, a) \;\middle|\; x = \bot \wedge a \in \mathcal{V} \vee x \in \mathcal{M} \wedge x \in \mathcal{V} \wedge a \preccurlyeq x \right\}$$

We write $\bullet m$ for (m, ε) and $\circ n$ for $(\bot, n)$. ■

Exercise 7.46. Show that the resource algebra $\mathcal{M}$ we used in the counter specification in Exercise 7.43 is exactly the resource algebra $\textsc{Auth}(\mathbb{N}_{\max})$ where $\mathbb{N}_{\max}$ is the resource algebra with carrier the natural number and operation the maximum. Its core is the identity function and all elements are valid. ◊

Exercise 7.47. Show the following properties of the resource algebra $\textsc{Auth}(\mathcal{M})$ for an arbitrary unital resource algebra $\mathcal{M}$.

- $\textsc{Auth}(\mathcal{M})$ is unital with unit $(\bot, \varepsilon)$, where ε is the unit of $\mathcal{M}$
- $\bullet x \cdot \bullet y \notin \mathcal{V}_{\textsc{Auth}(\mathcal{M})}$ for any x and y
- $\circ x \cdot \circ y = \circ(x \cdot y)$
- $\bullet x \cdot \circ y \in \mathcal{V} \Rightarrow y \preccurlyeq x$
- if $x \cdot z$ is valid in $\mathcal{M}$ then

  $$\bullet x \cdot \circ y \rightsquigarrow \bullet(x \cdot z) \cdot \circ(y \cdot z)$$

  in $\textsc{Auth}(\mathcal{M})$.

◊

Recall the isCounter predicate we used previously

$$\text{isCounter}(\ell, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists \iota. \boxed{\exists m.\, \ell \mapsto m * \boxed{\bullet m}^{\gamma}}^{\iota}.$$

We wish to incorporate into it the fraction p indicating how much of the counter this thread owns. We do this as follows.

$$\text{isCounter}(\ell, n, \gamma, p) = \boxed{\circ(p, n)}^{\gamma} * \exists \iota. \boxed{\exists m.\, \ell \mapsto m * \boxed{\bullet(1, m)}^{\gamma}}^{\iota}.$$

Thus the invariant stores the exact value of the counter, and since it knows its exact value the fraction is 1. The assertion $\boxed{\circ(p, n)}^{\gamma}$ connects the actual value of the counter to the value that is known to the particular thread. Now, to be able to read the exact value of the counter when p is 1 we need the property that if $\bullet(1, m) \cdot \circ(1, n)$ is valid then $n = m$. Further, we need the property that if $\bullet(1, m) \cdot \circ(p, n)$ is valid then $m \geq n$. Finally, we wish to get $\text{isCounter}(\ell, n + k, \gamma, p + q) \dashv\vdash \text{isCounter}(\ell, n, \gamma, p) * \text{isCounter}(\ell, k, \gamma, q)$. The way to achieve all this is to take the resource algebra $\textsc{Auth}((\mathbb{Q}_{01} \times \mathbb{N})_?)$ where

- $\mathbb{Q}_{01}$ is the resource algebra of fractions from Example 7.22,
- $\mathbb{N}$ is the resource algebra of natural numbers with *addition* as the operation, and every element is valid,
- and $(\mathbb{Q}_{01} \times \mathbb{N})_?$ is the option resource algebra on the product of the two previous ones.

Exercise 7.48. Show the following properties of the resource algebra $(\mathbb{Q}_{01} \times \mathbb{N})_?$.

- $(p, n) \cdot (q, m) = (p + q, n + m)$
- $(p, n) \preccurlyeq (q, m)$ and $q \leq 1$ if and only if $p = 1$ and $q = 1$ and $n = m$ or $p \leq q < 1$ and $n \leq m$, where $\leq$ is the standard ordering on rational and natural numbers.

And show the following properties of the resource algebra $\textsc{Auth}((\mathbb{Q}_{01} \times \mathbb{N})_?)$.

- $\circ(p,n) \cdot \circ(q,m) = \circ(p+q, n+m)$
- if $\bullet(1,m) \cdot \circ(p,n)$ is valid then $n \leq m$ and $p \leq 1$
- if $\bullet(1,m) \cdot \circ(1,n)$ is valid then $n = m$
- $\bullet(1,m) \cdot \circ(p,n) \rightsquigarrow \bullet(1,m+1) \cdot \circ(p,n+1)$.

◊

Using the results from the preceding exercise about ghost updates the following exercise is a straightforward adaptation of Exercise 7.43.

Exercise 7.49. Let isCounter be the predicate

$$\text{isCounter}(\ell,n,\gamma,p) = \boxed{\circ(p,n)}^{\gamma} * \exists \iota.\, \boxed{\exists m.\, \ell \mapsto m * \boxed{\bullet(1,m)}^{\gamma}}^{\iota}.$$

First show that for any p, q, n, and m, we have

$$\text{isCounter}(\ell, n+k, \gamma, p+q) \dashv\vdash \text{isCounter}(\ell,n,\gamma,p) * \text{isCounter}(\ell,k,\gamma,q).$$

Next show the following specifications for the methods defined above.

$$\{\mathsf{True}\}\ \text{newCounter}()\ \{u.\exists \gamma.\ \text{isCounter}(u,0,\gamma,1)\}$$
$$\forall p.\, \forall \gamma.\, \forall v.\, \forall n.\, \{\text{isCounter}(v,n,\gamma,p)\}\ \text{read}\, v\ \{u.u \geq n\}$$
$$\forall \gamma.\, \forall v.\, \forall n.\, \{\text{isCounter}(v,n,\gamma,1)\}\ \text{read}\, v\ \{u.u = n\}$$
$$\forall p.\, \forall \gamma.\, \forall v.\, \forall n.\, \{\text{isCounter}(v,n,\gamma,p)\}\ \text{incr}\, v\ \{u.u = () * \text{isCounter}(v,n+1,\gamma,p)\}$$

◊

Using the specification from the previous exercise we can now revisit Exercise 7.44 to give it the most precise specification.

Exercise 7.50. Let e be the program

$$\mathsf{let}\, c = \text{newCounter}()\, \mathsf{in}\, (\text{incr}\, c \| \text{incr}\, c); \text{read}\, c.$$

Using the specification of the counter module from the preceding exercise show the following specification for e.

$$\{\mathsf{True}\}\, e\, \{v.v = 2\}.$$

◊

7.8 Case Study: Invariants for Sequential Programs

Recursion through the store example.

8 First steps towards the base logic

The logic introduced thus far is powerful and can be used to verify very many examples of tricky concurrent algorithms. However there are constructs in the logic, *e.g.*, Hoare triples, which are responsible for many different reasoning principles, and as such have complex rules involving many different constructs at once, *e.g.*, the rules Ht-inv-open and Ht-frame-atomic. In this section we take first steps towards "logical simplication" – eventually (Section 11) we will see that all of Iris can be defined from a small core logic, which we refer to as the "Iris base logic". That is, of course, not only important for understanding and semantic modelling (Section 12), but also for building foundational tools for interactive verification in Iris. The simplifications described in this section suffice for *using* the Coq implementation of Iris (Section 9).

8.1 Weakest precondition

We start off by reducing the notion of a Hoare triple to that that of a *weakest precondition* assertion, which decouples the program from the precondition. Thus the weakest precondition is the minimal connection between the operational semantics of the program and its logical properties. It will turn out that, especially when using the Iris logic in the Coq proof assistant, it is often easier and more direct to use the weakest precondition assertion instead of (derived) Hoare triples.

Without further hesitation here is the typing rule for the new assertion.

$$\frac{\mathcal{E} \subseteq \mathsf{InvName} \qquad \Gamma \vdash e : \mathit{Exp} \qquad \Gamma \vdash \Phi : \mathit{Val} \to \mathsf{Prop}}{\Gamma \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} : \mathsf{Prop}}$$

In the same way that we write $v.Q$ in postconditions of Hoare triples we will write $v.Q$ instead of $\lambda v.Q$ in $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.Q\}$.

The intended meaning of the weakest precondition becomes clearer when we define Hoare triples in terms of it as

$$\{P\}\, e\, \{\Phi\}_{\mathcal{E}} \triangleq \Box (P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}).$$

Thus, $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ is indeed the *weakest* (*i.e.*, implied by any other) precondition such that e runs safely and if it terminates with a value v, the assertion $\Phi(v)$ holds. Further, the use of the $\Box$ modality is crucial. It guarantees that all the non-persistent resources required by e are contained in P.

This is consistent with the reading of Hoare triples explained in Section 7.2, where we explained that the resource required to run e are either in the precondition P, or owned by invariants, and invariants are persistent assertions.

The basic rules of this new assertion are listed in Figure 10 on page 81. The first part of the figure are basic structural rules. The rule wp-mono is analogous to the rule of consequence for Hoare triples, whereas the rule wp-frame is analogous to the frame rule Ht-frame, and the rule wp-frame-step is analogous to the rule Ht-frame-atomic. In fact, these rules for weakest precondition are used to derive the corresponding rules for Hoare triples. Next we have the expected rule wp-val, and the very important rule wp-bind which, analogously to the rule Ht-bind allows one to deconstruct the term into an evaluation context and a basic term for which we can use one of the basic rules for the weakest precondition assertion.

The rules for basic language constructs are stated in a style akin to the continuation passing style of programs, with an arbitrary postcondition Φ. This style allows for easy symbolic execution of programs, and circumvents the constant use of the rules wp-mono and wp-frame. To

see why this is so let us look at an alternative formulation of WP-ALLOC. This formulation is much closer to the Hoare triple rule HT-ALLOC.

Example 8.1. The rule

$$\frac{}{\triangleright(\ell \hookrightarrow v) \vdash \mathsf{wp}\, !\ell\, \{u.u = v * \ell \hookrightarrow v\}}\ \textsc{wp-load-direct}$$

is equivalent to the rule WP-LOAD.

First we derive WP-LOAD from WP-LOAD-DIRECT. Assuming WP-LOAD-DIRECT we have by WP-FRAME-STEP

$$\begin{aligned}\triangleright(\ell \hookrightarrow v) * \triangleright(\ell \hookrightarrow v \mathrel{-\!\!*} \Phi(v)) &\vdash \mathsf{wp}\, !\ell\, \{u.u = v * \ell \hookrightarrow v\} * \triangleright(\ell \hookrightarrow v \mathrel{-\!\!*} \Phi(v)) \\ &\vdash \mathsf{wp}\, !\ell\, \{u.u = v * \ell \hookrightarrow v * (\ell \hookrightarrow v \mathrel{-\!\!*} \Phi(v))\}\end{aligned}$$

which by WP-MONO yields $\mathsf{wp}\, !\ell\, \{\Phi\}$.

As you can see, the derivation required the use of WP-FRAME-STEP and WP-MONO. If we were to use the rule WP-LOAD-DIRECT in the proofs we would have to use these two structural rules constantly, which is tedious.

The converse derivation is straightforward. Assuming WP-LOAD we have

$$\begin{aligned}\triangleright(\ell \hookrightarrow v) &\vdash \triangleright(\ell \hookrightarrow v) * \triangleright(\ell \hookrightarrow v \mathrel{-\!\!*} (v = v * \ell \hookrightarrow v)) \\ &\vdash \mathsf{wp}\, !\ell\, \{u.u = v * \ell \hookrightarrow v\}\end{aligned}$$

where in the last step we used the rule WP-LOAD with $\Phi(u)$ being $u = v * \ell \hookrightarrow v$. ■

Exercise 8.2. Suppose we only had Hoare triples as a primitive in the logic, and we did not have the weakest precondition assertion. It turns out we can define, in the logic, an assertion $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.Q\}$ which satisfies the rules in Figure 10 as follows.

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq \exists P.\, P * \{P\}\, e\, \{\Phi\}_{\mathcal{E}}.$$

- Show that $\{P\}\, e\, \{\Phi\}_{\mathcal{E}} * P$ entails $\mathsf{wp}_{\mathcal{E}}\, E\, \{\Phi\}$, *i.e.*, if the Hoare triple $\{P\}\, e\, \{\Phi\}_{\mathcal{E}}$ holds then the precondition P implies $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$, *i.e.*, show the following entailment

$$\{P\}\, e\, \{\Phi\}_{\mathcal{E}} \vdash P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

- Show the rules in Figure 10 for $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ as defined here from the rules for Hoare triples described in the preceding sections. ◊

This exercise shows that the notions of Hoare triples and weakest preconditions are, at least with respect to the rules in Figure 10 and analogous rules for Hoare triples, essentially equivalent. The weakest precondition is the more minimal of the two, however, since it factors out the precondition. Further, we shall see in the next sections that some of the interactions with invariants can be more easily stated for the weakest precondition assertion. This leads to smaller, more manageable and principal rules.

Finally, notice that the rules in Figure 10 do not support working with invariants. Opening and closing of invariants is an operation that is of independent interest, *e.g.*, the ability to open and close invariants independently of Hoare triples is needed to define the concept of *logically atomic triples*[5], so it should not be tied to Hoare triples of the weakest precondition assertion directly. To support it we introduce a new concept, the *fancy update modality*.

[5] Logically atomic triples allow some reasoning principles, such as opening of invariants, also around programs which are "logically atomic", *e.g.*, they use locks, but are not atomic in the sense that they evaluate to a value in a single execution step.

Structural rules.

WP-MONO

$$\overline{(\forall v.\, \Phi(v) \mathrel{-\!\!*} \Psi(v)) * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Psi\}}$$

WP-FRAME

$$\overline{P * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{P * \Phi\}}$$

WP-FRAME-STEP

$$\frac{e \notin \mathit{Val}}{\triangleright P * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{P * \Phi\}}$$

WP-VAL

$$\overline{\Phi(v) \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\}}$$

WP-BIND

$$\overline{\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \mathsf{wp}_{\mathcal{E}}\, E[\, v\,]\, \{\Phi\}\} \vdash \mathsf{wp}_{\mathcal{E}}\, E[\, e\,]\, \{\Phi\}}$$

Rules for basic language constructs.

WP-FORK

$$\overline{\triangleright \Phi() * \triangleright \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \mathsf{True}\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{fork}\, \{e\}\, \{\Phi\}}$$

WP-ALLOC

$$\overline{\triangleright (\forall \ell.\, \ell \hookrightarrow v \mathrel{-\!\!*} \Phi(\ell)) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{ref}(v)\, \{\Phi\}}$$

WP-LOAD

$$\overline{\triangleright (\ell \hookrightarrow v) * \triangleright (\ell \hookrightarrow v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}_{\mathcal{E}}\, !\ell\, \{\Phi\}}$$

WP-STORE

$$\overline{\triangleright (\ell \hookrightarrow v) * \triangleright (\ell \hookrightarrow w \mathrel{-\!\!*} \Phi()) \vdash \mathsf{wp}_{\mathcal{E}}\, (\ell \leftarrow w)\, \{\Phi\}}$$

WP-CAS-SUC

$$\overline{\triangleright (\ell \mapsto v) * \triangleright (\ell \mapsto w \mathrel{-\!\!*} \Phi(\mathsf{true})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{cas}(\ell, v, w)\, \{\Phi\}}$$

WP-CAS-FAIL

$$\overline{v \neq v' \wedge \triangleright (\ell \mapsto v) * \triangleright (\ell \mapsto v \mathrel{-\!\!*} \Phi(\mathsf{false})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{cas}(\ell, v', w)\, \{\Phi\}}$$

WP-REC

$$\overline{\triangleright \mathsf{wp}_{\mathcal{E}}\, e[v/x][(\mathsf{rec}\, f(x) = e)/f]\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, (\mathsf{rec}\, f(x) = e) v\, \{\Phi\}}$$

WP-PROJ

$$\overline{\triangleright \mathsf{wp}_{\mathcal{E}}\, v_i\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \pi_i\, (v_1, v_2)\, \{\Phi\}}$$

WP-IF-TRUE

$$\overline{\triangleright \mathsf{wp}_{\mathcal{E}}\, e_1\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{if}\, \mathsf{true}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \{\Phi\}}$$

WP-IF-FALSE

$$\overline{\triangleright \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{if}\, \mathsf{false}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\, \{\Phi\}}$$

WP-MATCH

$$\overline{\triangleright \mathsf{wp}_{\mathcal{E}}\, e_i\, [u/x_i]\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{match}\, \mathsf{inj}_i\, u\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end}\, \{\Phi\}}$$

Figure 10: Rules for the weakest precondition assertion.

8.2 Fancy update modality

The fancy update modality allows us to get resources out of knowledge that an invariant exists, *i.e.*, to get P from $\boxed{P}^{\iota}$, and to put resources back into an invariant, *i.e.*, to close the invariant. As we explained in Section 7.2 invariants are persistent, in particular duplicable. Thus we cannot simply get resources out of invariants in the sense of the rule $\boxed{P}^{\iota} \vdash P$ or $\boxed{P}^{\iota} \vdash \triangleright P$; this would lead to inconsistency. We need to keep track of the fact that we were allowed to open this particular invariant, and that we are not allowed to open this particular invariant again until we have closed it. Thus, the rule for opening invariants will be

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P}$$

where ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}$ is the *fancy update modality*, and $\mathcal{E}_1$ and $\mathcal{E}_2$ are masks, *i.e.*, sets of invariant names (cf. Section 7.2).

The intuition behind the modality ${}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P$ is that it contains resources r which, together with resources in invariants named $\mathcal{E}_1$, can be updated (via frame preserving update) to resources which can be split into resources satisfying P and resources in invariants named $\mathcal{E}_2$. Thus in particular the fancy update modality subsumes the update modality $\Rrightarrow$ introduced in Section 7, in the sense that $\Rrightarrow P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P$, *i.e.*, if the set of invariant names available does not change. The rules for the fancy update modality are listed in Figure 11. We describe the rules now, apart from the rule Fup-timeless, which we describe in the next section, when we introduce the notion of timelessness.

Fup-mono
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q}$$

Fup-intro-mask
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} P}$$

Fup-trans
$$\frac{}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} P}$$

Fup-frame
$$\frac{\mathcal{E}_f \text{ disjoint from } \mathcal{E}_1 \cup \mathcal{E}_2}{Q * {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P)}$$

Fup-upd
$$\frac{}{\Rrightarrow P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P}$$

Fup-timeless
$$\frac{\vdash P \text{ timeless}}{\triangleright P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P}$$

Inv-alloc
$$\frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_2} \exists \iota \in \mathcal{E}_1.\, \boxed{P}^{\iota}}$$

Inv-open
$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright P * \left(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right)}$$

Figure 11: Basic rules for the fancy update modality.

Introduction and structural rules of the fancy update modality The following rules are analogous to the rules for the update modality introduced in Section 7.2.

Fup-mono
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q}$$

Fup-intro-mask
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} P}$$

Fup-trans
$$\frac{}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} P}$$

The rule Fup-intro-mask is perhaps a bit surprising since it introduces two instances of the fancy update modality, with swapped masks. This generality is useful since, in general, we do not

have $P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$. Indeed, if, for example, $P \vdash {}^{\emptyset}\!\Rightarrow^{\{\iota\}} P$ was provable it would mean that any resource in P could be split into a resource satisfying the invariant named ι, and a resource satisfying P. This cannot hold in general, of course. However, using Fup-trans together with Fup-intro-mask, we can derive the following introduction rule where the masks are the same:

Fup-intro

$$\frac{}{P \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P}$$

We will write $\Rightarrow_{\mathcal{E}} P$ for ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P$.

Next we have a rule relating the modality with separating conjunction, analogous to upd-frame, but in addition to framing of resources, we can also frame on additional invariant names.

Fup-frame

$$\frac{\mathcal{E}_f \text{ disjoint from } \mathcal{E}_1 \cup \mathcal{E}_2}{Q * {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P)}$$

Perhaps the rule looks daunting. The following derived rules are perhaps more natural, since they only manipulate a single concept (either the frame, or the masks) at a time.

$$\frac{}{Q * {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} (Q * P)} \qquad \frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{\Rightarrow_{\mathcal{E}_1} P \vdash \Rightarrow_{\mathcal{E}_2} P}$$

Exercise 8.3. Derive the above two rules from Fup-frame. ◊

Next we have the rule relating fancy update modality with the ordinary update modality. The rule states that the fancy update modality is logically weaker than the update modality.

Fup-upd

$$\frac{}{\Rightarrow P \vdash \Rightarrow_{\mathcal{E}} P}$$

Note that in combination with the previous rules for ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2}$, the rules Ghost-alloc and Ghost-update remain valid if we replace $\Rightarrow$ with $\Rightarrow_{\mathcal{E}}$, for any mask $\mathcal{E}$.

Fancy update modality and invariants Finally, we have rules for allocation and opening of invariants:

Inv-alloc

$$\frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_2} \exists \iota \in \mathcal{E}_1. \boxed{P}^{\iota}}$$

Inv-open

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright P * \left(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rightarrow^{\mathcal{E}} \mathsf{True}\right)\right)}$$

The allocation rule should not be surprising, perhaps apart from the two different sets of invariant names. An intuitive reason for why the two sets $\mathcal{E}_1$ and $\mathcal{E}_2$ of invariant names are not required to be related is that we only allocate a new invariant – the mask $\mathcal{E}_2$ has to do with opening and closing of invariants, as can be seen in Inv-open.

An equivalent rule to Inv-alloc is the following

Inv-alloc-empty

$$\frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash {}^{\emptyset}\!\Rightarrow^{\emptyset} \exists \iota \in \mathcal{E}_1. \boxed{P}^{\iota}}$$

Exercise 8.4. Derive INV-ALLOC from INV-ALLOC-EMPTY. ◊

The rule INV-OPEN is used not just to open invariants, but also to close them. It implies the following two rules

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P} \qquad \frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)}$$

The first one of which is the pure invariant opening rule. It states that we can get resources out of an invariant, but only *later*. Removing the later from the rule would be unsound, as we will see in Section 11. The second rule is the invariant closing rule. It shows how resources can be transferred back into invariants. The crucial parts in this rule are the invariant masks. In particular, the assertion ${}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$ is *not* equivalent to True. It contains only those resources which can be combined with resources in invariants named $\mathcal{E} \setminus \{\iota\}$ to get resources in invariants named $\mathcal{E}$, *i.e.*, it contains the resources in the invariant named ι.

Exercise 8.5. Show the following property of the fancy update modality.

$$ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} (P \mathrel{-\!\!*} Q) \vdash P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$$

◊

8.2.1 The fancy update modality and weakest precondition

Finally, we have rules connecting the new update modality to the weakest precondition assertion, and thus to Hoare triples and program specfications. These rules generalise several of the rules we have seen before. In particular HT-INV-ALLOC, HT-INV-OPEN and the previous rule HT-CSQ will be derivable from the rules introduced in this section.

The rules for the relationship between the fancy update modality and weakest preconditions are listed in Figure 12. The rule WP-VUP states that we can remove the update modalities around

WP-VUP
$$\frac{}{\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{v. \Rrightarrow_{\mathcal{E}} \Phi(v)\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

WP-ATOMIC
$$\frac{e \text{ is an atomic expression}}{{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \{v. {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} \Phi(v)\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

WP-FRAME-STEP
$$\frac{e \notin \mathit{Val} \qquad \mathcal{E}_2 \subseteq \mathcal{E}_1}{\left({}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \triangleright {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} P\right) * \mathsf{wp}_{\mathcal{E}_2}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{P * \Phi\}}$$

Figure 12: Rules connecting fancy view shifts to the weakest precondition assertion.

and inside the weakest precondition assertion. This is important because in general we do not have $\Rrightarrow_{\mathcal{E}} P \vdash P$, and so proving $\Rrightarrow_{\mathcal{E}} P$ is weaker than proving P. The rule WP-VUP states that this is not the case for the weakest precondition assertion. We can use this rule to, for example, do frame preserving updates inside the weakest precondition assertion.

Exercise 8.6. Derive the following rule.

$$\frac{a \rightsquigarrow b}{\mathsf{wp}_{\mathcal{E}}\, e\, \{v. \Phi(v) * \boxed{a}^{\gamma}\} \vdash \mathsf{wp}\, e\, \{v. \Phi(v) * \boxed{b}^{\gamma}\}}$$

◊

Next is the rule WP-ATOMIC, which is similar to the rule HT-INV-OPEN. It is crucial here that e is an atomic expression. If it was not then a similar counterexample as the one for the rule HT-INV-OPEN, which is explained in Example 7.7, would apply, and the weakest precondition assertion would not be sound for the operational semantics of the language. The rule is very general, so let us see how it allows us to recover some rules for working with invariants.

Example 8.7. Let $\mathcal{E}$ be a set of invariant names and $\iota \in \mathcal{E}$ and *e an atomic expression*. We derive the following rule for accessing invariants using the weakest precondition assertion.

$$\frac{\text{WP-INV-OPEN}\quad e \text{ is an atomic expression}}{\boxed{I}^{\iota} * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}}$$

We have

$$\begin{aligned}
\boxed{I}^{\iota} * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) &\vdash \left({}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright I * \left(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right)\right) * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) && \text{(INV-OPEN)}\\
&\vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\left(\triangleright I * \left(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right) * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right)\right) && \text{(FUP-FRAME)}\\
&\vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\left(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right) * \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) && (\mathrel{-\!\!*}\text{E})\\
&\vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\left(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right) * (\triangleright I * \Phi(v))\} && \text{(WP-FRAME)}\\
&\vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\left({}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right) * \Phi(v)\} && \text{(WP-MONO and } \mathrel{-\!\!*}\text{E)}\\
&\vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\left({}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \mathsf{True} * \Phi(v)\right)\} && \text{(FUP-FRAME)}\\
&\vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\left({}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}} \Phi(v)\right)\} && \text{(FUP-MONO)}\\
&\vdash \mathsf{wp}\, e\,\{\Phi\} && \text{(WP-ATOMIC)}
\end{aligned}$$

■

The rule derived in the preceding example can be somewhat strengthened.

Exercise 8.8. Derive the following rules.

$$\left({}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \boxed{I}^{\iota}\right) * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \tag{30}$$

$${}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \left(\boxed{I}^{\iota} * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right)\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \tag{31}$$

$$\boxed{I}^{\iota} * {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \tag{32}$$

$$\boxed{I}^{\iota} * \left(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}\!\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \tag{33}$$

◊

As we mentioned above the rule WP-ATOMIC is similar to the rule HT-INV-OPEN. In fact, the latter is derivable from the rule we derived in Example 8.7, as we now demonstrate.

Example 8.9 (Derivation of HT-INV-OPEN from WP-INV-OPEN)**.** Let e be an atomic expression. We are to show

$$\frac{\text{HT-INV-OPEN}\quad S \wedge \boxed{I}^{\iota} \vdash \{\triangleright I * P\}\, e\, \{v.\triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}}}{S \wedge \boxed{I}^{\iota} \vdash \{P\}\, e\, \{\Phi\}_{\mathcal{E}}}$$

recalling that we have defined $\{P\}\,e\,\{\Phi\}_{\mathcal{E}}$ as $\Box(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\})$. Let us show it. Since invariants and Hoare triples are persistent we have

$$\begin{aligned} S \wedge \boxed{I}^{\iota} &\vdash (S \wedge \boxed{I}^{\iota}) \wedge \boxed{I}^{\iota} \\ &\vdash (\{\triangleright I * P\}\, e\,\{v.\triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}}) \wedge \boxed{I}^{\iota} \\ &\vdash \Box\Big(\{\triangleright I * P\}\, e\,\{v.\triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}} * \boxed{I}^{\iota}\Big) \end{aligned}$$

and thus it suffices to show

$$\{\triangleright I * P\}\, e\,\{v.\triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}} * \boxed{I}^{\iota} \vdash P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}.$$

by ALWAYS-MONO. In fact by ALWAYS-E it suffices to show

$$\Big(\triangleright I * P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\Big) * \boxed{I}^{\iota} \vdash P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}$$

which is equivalent to showing

$$\Big(\triangleright I * P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\Big) * \boxed{I}^{\iota} * P \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}$$

by the wand introduction rule. Now

$$\Big(\triangleright I * P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\Big) * \boxed{I}^{\iota} * P \vdash \Big(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\,\{v.\triangleright I * \Phi(v)\}\Big) * \boxed{I}^{\iota}$$

by the wand elimination rule, which in turn yields

$$\mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\}$$

by WP-INV-OPEN derived in Example 8.7. ■

The final rule is WP-FRAME-STEP. This is analogous to the rule HT-FRAME-ATOMIC, which allows us to remove laters from frames in the precondition, provided the term is atomic. Here, the term is not required to be atomic, but it is important that it is not a value. The fancy update modalities included in the rule are useful in certain cases, thus the rule is stated in full generality.

Exercise 8.10. Derive the following rules from WP-FRAME-STEP.

$$\frac{e \notin \mathit{Val}}{\triangleright P * \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\,\{v.P * \Phi(v)\}} \qquad \frac{e \notin \mathit{Val} \qquad S \vdash \{P\}\, e\,\{v.Q\}_{\mathcal{E}}}{S \vdash \{P * \triangleright R\}\, e\,\{v.Q * R\}_{\mathcal{E}}}$$

To derive the first rule, recall that $P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P$ for any P and any mask $\mathcal{E}$. ◊

Finally, note that there is no special rule needed for allocating invariants in connection with weakest preconditions. This is in contrast to Hoare triples, where allocating an invariant means transferring resources from the *precondition* to the invariant. With weakest preconditions allocation of invariants is handled separately, and interaction of invariants and weakest preconditions is governed by the fancy update modality.

Fancy view shifts Finally, we define the *fancy view shift* $P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q$ from the fancy update modality as

$$P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \triangleq \Box(P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q).$$

If $\mathcal{E}_1 = \mathcal{E}_2$ we write $P \Rrightarrow_{\mathcal{E}_1} Q$ for $P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_1} Q$. This concept is analogous to how view shifts are defined from the update modality in Section 7. The concept makes it easier to state some of the (derived) rules involving Hoare triples.

Exercise 8.11. Derive the following rules for the fancy view shift.

- $$\textsc{Fvs-refl}\ \frac{}{\cdot \vdash P \Rrightarrow_{\mathcal{E}_1} P} \qquad \textsc{Fvs-trans}\ \frac{S \vdash P \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \qquad S \vdash Q \ {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} R}{S \vdash P \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} R}$$

- $$\textsc{Fvs-imp}\ \frac{S \vdash \Box(P \Rightarrow Q)}{S \vdash P \Rrightarrow_{\mathcal{E}} Q} \qquad \textsc{Fvs-wand}\ \frac{S \vdash \Box(P \mathrel{-\!\!*} Q)}{S \vdash P \Rrightarrow_{\mathcal{E}} Q}$$

- $$\textsc{Fvs-frame}\ \frac{S \vdash P \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q}{S \vdash P * R \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q * R} \qquad \textsc{Fvs-mask-frame}\ \frac{S \vdash P \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \qquad (\mathcal{E}_1 \cup \mathcal{E}_2) \cap \mathcal{E}_f = \emptyset}{S \vdash P * R \ {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} Q * R}$$

- $$\textsc{Fvs-timeless}\ \frac{\vdash P \text{ timeless}}{\cdot \vdash \triangleright P \Rrightarrow_{\mathcal{E}} P}$$

- $$\textsc{Fvs-alloc-I}\ \frac{\mathcal{E} \text{ infinite}}{\cdot \vdash \triangleright P \ {}^{\emptyset}\!\Rrightarrow^{\emptyset} \exists \iota \in \mathcal{E}. \boxed{P}^{\iota}} \qquad \textsc{Fvs-open-I}\ \frac{}{\boxed{P}^{\iota} \vdash \mathsf{True} \ {}^{\{\iota\}}\!\Rrightarrow^{\emptyset} \triangleright P}$$

◊

Hoare triples and fancy view shifts With the new concepts we can present the final generalisation of the rules for Hoare triples. The most general rule of consequence we consider is the following

$$\textsc{Ht-csq}\ \frac{S \vdash P' \ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P \qquad S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}} \qquad S \vdash \forall v.\, Q(v) \ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} Q'(v)}{S \vdash \{P'\}\, e\, \{v.Q'\}_{\mathcal{E}}}$$

From now on Ht-csq will refer to this instance.

Exercise 8.12. Derive the above rule of consequence. ◊

The next rule is a generalisation of Ht-frame-atomic.

$$\textsc{Ht-frame-step}\ \frac{e \notin \mathit{Val} \qquad S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}_2} \qquad S \vdash R_1 \ {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \triangleright R_2 \qquad S \vdash R_2 \ {}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} R_3 \qquad \mathcal{E}_2 \subseteq \mathcal{E}_1}{S \vdash \{P * R_1\}\, e\, \{v.Q * R_3\}_{\mathcal{E}_1}}$$

It allows us to remove the later modality from the frame in cases where the term e is not a value. The side-condition in the rule corresponds to the side-condition in the rule wp-frame-step.

Exercise 8.13. Derive the rule Ht-frame-step from the rule wp-frame-step. ◊

8.3 Timeless propositions

We have already mentioned *timeless* propositions in the previous section. One of the rules for the fancy update modality is the rule Fup-timeless

$$\frac{\text{Fup-timeless}}{\vdash P \text{ timeless}}{\triangleright P \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} P}$$

which allows allows us to remove a later provided the proposition P is timeless, and the conclusion is under the fancy update modality. Note that if we wanted $\triangleright P \vdash P$ for timeless propositions then the only timeless proposition would be True. This follows from the Löb induction principle.

Now, what exactly is a timeless proposition? Recall the intuition behind the later modality. The proposition $\triangleright P$ holds if P holds in the future. Now, some propositions do not depend on time. For example, if n and m are natural numbers then $n = m$ is either always true, or always false. These are the propositions which we call timeless.

Definition 8.14. A proposition P is timeless if the following entailment holds

$$\triangleright P \vdash P \vee \triangleright \mathsf{False}$$

We write

$$\vdash P \text{ timeless}$$

for the judgement stating that P is timeless, or

$$\Gamma \vdash P \text{ timeless}$$

if the variable context Γ is important. ■

There is a perhaps curious $\triangleright\mathsf{False}$ appearing in the definition. In order to have the powerful Löb induction rule we must have that if $\triangleright P \vdash P$, then P is necessarily equivalent to True. Semantically, this means there has to be a "final time", where there is no future. In order for $\triangleright P$ to be well-defined it must be that $\triangleright P$ holds in this final world. The proposition $\triangleright\mathsf{False}$ is a proposition which holds exactly at this final time, but does not hold otherwise.

All ordinary propositions are timeless. By ordinary propositions we mean such things as equality on all base types apart from Prop, basic relations such as $\leq$ or $\geq$ on natural numbers and so on. Moreover being timeless is preserved by almost all the constructs of logic, as stated in Figure 13. A general guiding principle we can discern from these rules is that if a predicate does not involve a later or update modality, or an arbitrary predicate P, then it is timeless.

Properties of timeless propositions In the examples in the preceding sections we have seen that opening invariants leads to some complications with the later modality. When opening the invariant $\boxed{P}^{\iota}$ we only get the proposition $\triangleright P$ in the precondition of the Hoare triple, as opposed to P. We worked around this by using Ht-frame-atomic together with stronger rules for Hoare triples from Section 5.1, but this is often quite inconvenient, especially since we often need to remove $\triangleright$ from propositions which, intuitively, do not depend on time.

The essence of why timelessness is a useful property is captured by the following rule, which relates timeless propositions to Hoare triples.

$$\frac{\text{Ht-timeless-pre-post}}{\vdash P_1 \text{ timeless} \qquad \vdash Q_1 \text{ timeless} \qquad \{P_1 * P_2\}\, e\, \{v. \triangleright Q_1 * Q_2\}}{\{\triangleright P_1 * P_2\}\, e\, \{v. Q_1 * Q_2\}}$$

$$\frac{}{\vdash \mathsf{True}\ \mathsf{timeless}} \qquad \frac{}{\vdash \mathsf{False}\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \lor Q\ \mathsf{timeless}}$$

$$\frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \land Q\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \Rightarrow Q\ \mathsf{timeless}}$$

$$\frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P * Q\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \mathrel{-\!\!*} Q\ \mathsf{timeless}} \qquad \frac{\Gamma, x : \tau \vdash \Phi\ \mathsf{timeless}}{\Gamma \vdash \forall x.\, \Phi\ \mathsf{timeless}}$$

$$\frac{\Gamma, x : \tau \vdash \Phi\ \mathsf{timeless}}{\Gamma \vdash \exists x.\, \Phi\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless}}{\vdash \Box P\ \mathsf{timeless}}$$

Moreover ghost ownership is timeless.

$$\frac{\textsc{Ghost-RA-timeless} \quad M \text{ is a resource algebra} \qquad a \in M}{\vdash \boxed{a}^{\gamma}\ \mathsf{timeless}}$$

Figure 13: Rules for timeless propositions.

The rule states that we can remove one $\triangleright$ from pre- and postconditions provided the propositions are timeless. Note that there is no restriction on the expression e being atomic, as there is in Ht-frame-atomic. However Ht-timeless-pre-post is in general incomparable with Ht-frame-atomic since the latter applies to arbitrary frames P, not just to timeless ones.

Exercise 8.15. Derive the rule Ht-timeless-pre-post from the generalized rule of consequence involving the fancy view shifts introduced in the previous section. ◊

Exercise 8.16. Derive the following rules for timeless propositions.

$$\frac{\textsc{Ht-timeless-pre} \quad \vdash P_1\ \mathsf{timeless} \qquad \{P_1 * P_2\}\, e\, \{v.Q\}}{\{\triangleright P_1 * P_2\}\, e\, \{v.Q\}} \qquad \frac{\textsc{Ht-timeless-post} \quad \vdash Q_1\ \mathsf{timeless} \qquad \{P\}\, e\, \{v. \triangleright Q_1 * Q_2\}}{\{P\}\, e\, \{v.Q_1 * Q_2\}}$$

◊

In the examples we have done thus far the new rules would not help to reduce complexity noticeably. Later on, however, we will see that it is crucial for examples involving complex ghost state and invariants. Moreover, when using Iris in Coq it simplifies its use significantly, since tactics can automatically derive that propositions are timeless, and thus automatically remove the later modality in many places. The reason it was not needed until now is that we have strong rules for Hoare triples in the sense that, for basic stateful operations, it suffices to have $\triangleright(\ell \hookrightarrow v)$ in the precondition. Typically, when using invariants we will get such an assertion after opening an invariant. But when we use more complex ghost state, then we shall get propositions of the form $\triangleright\boxed{a}^{\gamma}$ in the precondition. Using such is difficult without timelessness.

A conceptual reason for why timelessness is a useful and needed concept is the following. Iris supports nested and higher-order invariants. For this reason it is crucial, as we shall see later on, that when opening an invariant we do not get access to the resources *now*, but only

later, *i.e.*, opening an invariant gives us $\triangleright P$ in the precondition. However this is only needed if P refers to other invariants, or is a higher-order predicate. For first order-predicates, which do not refer to other invariants, it is safe to get the resources immediately. Using the notion of timelessness, we can recover some of the convenience of logics which support only first-order, predicative invariants, but retain the ability to form and use higher-order invariants.

Exercise 8.17. Derive the following rules for invariant opening. We assume $\mathcal{E}$ is a set of invariant names and $\iota \in \mathcal{E}$.

$$\frac{\textsc{wp-inv-timeless}\quad e \text{ is an atomic expression} \qquad \vdash I \text{ timeless}}{\boxed{I}^{\iota} * \left(I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

$$\frac{\textsc{Ht-inv-timeless}\quad e \text{ is an atomic expression} \qquad \vdash I \text{ timeless} \qquad S \wedge \boxed{I}^{\iota} \vdash \{I * P\}\, e\, \{v. I * Q\}_{\mathcal{E}\setminus\{\iota\}}}{S \wedge \boxed{I}^{\iota} \vdash \{P\}\, e\, \{v. Q\}_{\mathcal{E}}}$$

$\Diamond$

The exercise establishes some properties of timeless assertions which are used implicitly when the Iris logic is used in the Coq proof assistant.

Exercise 8.18. Assuming P is timeless derive the following rules.

$$\frac{Q * P \vdash \Rrightarrow_{\mathcal{E}} R}{Q * \triangleright P \vdash \Rrightarrow_{\mathcal{E}} R} \qquad \frac{Q * P \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}{Q * \triangleright P \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

$\Diamond$

8.4 Invariant namespaces

Invariant namespaces are the final conceptual ingredient needed to use the Iris logic in the Coq proof assistant. They simplify the use of the logic when we need to open multiple invariants. Let us see why. Suppose we are proving

$$\boxed{P_1}^{\iota_1} \wedge \boxed{P_2}^{\iota_2} \vdash \{P\}\, e\, \{\Phi\}_{\mathcal{E}}.$$

Then to use invariants I_1 and I_2 at the same time we need to know $\iota_1, \iota_2 \in \mathcal{E}$ and that $\iota_1 \neq \iota_2$. The reason we need to know the last inequality is that after opening the first invariant we need to prove

$$\boxed{P_1}^{\iota_1} \wedge \boxed{P_2}^{\iota_2} \vdash \{\triangleright I_1 * P\}\, e\, \{v. \triangleright I_1 * \Phi(v)\}_{\mathcal{E}\setminus\{\iota_1\}}$$

and thus if ι_1 and ι_2 were the same, then we could not open them again. So how can we know that $\iota_1 \neq \iota_2$? The only way we can guarantee it is by using suitable sets of invariant names when allocating invariants. Recall (one variant of) the invariant allocation rule

$$\frac{\textsc{Inv-alloc}\quad \mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash \Rrightarrow_{\emptyset} \exists \iota \in \mathcal{E}_1.\, \boxed{P}^{\iota}}$$

We can *choose* an infinite set of invariant names from which ι is drawn. Hence we can use different sets in different parts of the proof in order to guarantee name inequalities. Invariant namespaces are used to denote these infinite sets of invariant names.

There are different ways to encode them, but one way to think of them is to think of invariant names as strings. An invariant namespace is then also a string $\mathcal{N}$, but it denotes the set of all strings whose prefix is $\mathcal{N}$. We write this set as $\mathcal{N}^\uparrow$. With this encoding, if $\mathcal{N}$ is a namespace, then, say, $\mathcal{N}$.lock and $\mathcal{N}$.counter are two other namespaces, and, importantly, they denote disjoint sets of invariant names, *i.e.*, the sets $(\mathcal{N}.\mathsf{lock})^\uparrow$ and $(\mathcal{N}.\mathsf{counter})^\uparrow$ are disjoint. To make use of namespaces we define some abbreviations. We define

$$\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}^\uparrow. \boxed{P}^{\iota}.$$

With this notation the invariant allocation rule looks simpler, as

$$\triangleright P \vdash \Rightarrow_{\emptyset} \boxed{P}^{\mathcal{N}}.$$

for any chosen namespace $\mathcal{N}$. Other rules for working with invariants need to change slightly to use $\boxed{P}^{\mathcal{N}}$. The rule for opening invariants becomes

$$\textsc{Inv-open-namespace}\quad \frac{\mathcal{N}^\uparrow \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\mathcal{N}^\uparrow} \left(\triangleright P * \left(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}^\uparrow}\!\Rightarrow^{\mathcal{E}} \mathsf{True}\right)\right)}$$

and the rule for opening invariants in connection with the weakest precondition assertion is thus

$$\textsc{wp-inv-open-namespace}\quad \frac{e \text{ is an atomic expression} \qquad \mathcal{N}^\uparrow \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}^\uparrow} e\{v. \triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}} e\{\Phi\}}$$

Unfortunately, with the rules we have presented thus far we cannot derive Inv-open-namespace from Inv-open. We will be able to do this once we *define* the fancy update modality in terms of other connectives in Section 11, but for now we remark that the rule is sound, and derivable from a general property of the fancy update modality stated in the following exercise.

Exercise 8.19. Assume the following property of the fancy update modality, for any masks $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_f$ such that $\mathcal{E}_1$ is disjoint from $\mathcal{E}_f$.

$${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} \left(P * (Q \mathrel{-\!\!*} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1} R)\right) \vdash {}^{\mathcal{E}_1\cup\mathcal{E}_f}\!\Rightarrow^{\mathcal{E}_2} \left(P * (Q \mathrel{-\!\!*} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1\cup\mathcal{E}_f} R)\right).$$

Use this property to derive Inv-open-namespace from Inv-open. ◊

To summarize, namespaces are a convenience feature for dealing with multiple invariants. They explicitly record the infinite set of invariant names we have chosen when allocating the invariant. The rules for using invariants can then be presented in such a way that they only mention the namespace, and not the concrete name the invariant has. Moreover namespaces have a tree-like structure, with disjoint children. These are called subnamespaces. For example, if $\mathcal{N}$ is a namespace then $\mathcal{N}$.lock and $\mathcal{N}$.counter are two disjoint subnamespaces. Hence if we allocate invariants $\boxed{P_1}^{\mathcal{N}.\mathsf{lock}}$ and $\boxed{P_2}^{\mathcal{N}.\mathsf{counter}}$, then it is immediately clear that we can open both of them at the same time. We do not need to keep track of additional name inequalities elsewhere in our context.

Invariant namespaces are well supported in the Iris proof mode in Coq. Hence, most of the time, when working with invariants, the side-conditions on masks are discharged automatically in the background This simplifies proofs and enables the user to focus on the interesting parts of the verification.

9 Iris Proof Mode in Coq

We give a brief introduction to how to use the Iris logic within the Coq proof assistant. The formalization is available at gitlab.mpi-sws.org/FP/iris-coq, where the reader can also find detailed installation instructions. The formalization of Iris in Coq has many parts. First, the semantics of the logic is formalized, and all the basic proof rules are proved sound with respect to this semantics. This is a significant formalization effort. On top of this formalization a number of derived rules and constructs are defined. The top layer of this formalization is the *interactive proof mode*. This proof mode is the most important part of the formalization to learn to be able to prove program specifications, and most of the other parts are hidden behind this layer of abstraction. However occasionally details of the model do leak through this abstraction, but we shall either explain those as we go along, or the reader will have to take them on faith until we explain the model in Section 12.

Instead of describing Coq proofs and tactics in this documents, which would be difficult to maintain up to date, we have provided two heavily commented files, which explain how the interactive proof mode can be used to prove program specifications on two examples. These files are available here. As a first example we prove specification from Example 7.5. This is the simplest non-trivial concurrent example. It uses invariants, but no ghost state. As a second example we prove counter specifications from Section 7.7, and using the precise counter specification we show a proof of the client from Exercise 7.50. This second example shows how to use all of the main features of Iris in Coq. In particular it shows how to use different resource algebras in Coq.

Furthermore, there are many other examples, and case studies in the iris-examples repository, which is publicly available at gitlab.mpi-sws.org/FP/iris-examples/.

Interactive proof mode Interactive proof mode is a set of tactics to manipulate judgements $\mathcal{S} \vdash Q$. For various reasons it has proved useful to split the context $\mathcal{S}$ into three parts. The first part are the pure facts, such as equality of values, comparison of natural numbers, *etc.*, the second part are the persistent Iris assertions, and the last part are general Iris assertions. So to be more precise, the interactive proof mode tactics manipulate such judgements and the tactics are aware, for instance, that the assertions in the persistent context can be duplicated. Let us see how this looks on an example proof. We are proving

$$\square P * Q \vdash (P * Q) * P.$$

The initial goal looks as follows.

```
Σ : gFunctors
P, Q : iProp Σ
--------------------------
□ P * Q -* (P * Q) * P
```

Ignoring Σ, which has to do with specifying which resource algebras are available, this is an ordinary Coq goal. The assumptions are that P and Q are Iris propositions, and the goal is to prove the entailment. Note that the $\vdash$ is replaced with $-\!*$, for reasons which are not important. It is simply different notation for the same thing.

We then enter the proof mode at which point our goal looks as follows.

```
Σ : gFunctors
P, Q : iProp Σ
============================
"HP" : P
------------------------------------□
"HQ" : Q
------------------------------------*
(P * Q) * P
```

The Coq context, above the double line, stays the same, but the goal is different. It consists of two contexts, and a conclusion. The first context contains one assumption P, named `HP`. Assumptions are named so that they can be referred to by tactics, analogously how This is the context of *persistent assumptions*. Every assumption in this context implicitly has an $\Box$ modality around it.

The second context also contains one assumption, Q, and the assumption has name `HQ`. This is a context of arbitrary Iris assertions.

To prove the goal, if we were using the rules of the logic directly, we would duplicate the assumption $\Box P$, and then use the separating conjunction introduction rule. There is a tactic which corresponds to the separating conjunction introduction rule, and the tactic knows that persistent assertions can be duplicated. Thus using this tactic we get the following two goals.

```
  Σ : gFunctors
  P, Q : iProp Σ
  ============================
  "HP" : P
  ------------------------------------□
  "HQ" : Q
  ------------------------------------*
  P * Q


subgoal 2 (ID 147) is:
 "HP" : P
------------------------------------□
P
```

Notice how in the first goal we have assertions P and Q available, whereas in the second we only have P available, since Q is not persistent.

In the accompanying Coq example files we explain how to use the tactics and manipulate contexts to achieve this.

Hoare triples in Iris Coq One point of difference of the Iris logic in Coq as opposed to the one presented in this paper is the definition of Hoare triples. Recall that we defined Hoare triples as

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\}).$$

In Coq they are defined slightly differently, using the similar mode of use of weakest precondition specifications with an arbitrary postcondition. To wit, they are defined as

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(\forall \Psi, P \mathrel{-\!\!*} \triangleright(\forall v, \Phi(v) \mathrel{-\!\!*} \Psi(v)) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Psi\}).$$

If there was no later modality the two definitions would be rather trivially equivalent. The reason for introducing the later modality is technical, and it is there purely for reasons of convenience.

Exercise 9.1. Show that for expressions *e which are not values* the two definitions are logically equivalent. ◊

Thus, the only place where they differ slightly is for values. But since these triples are in practice never used for values, they are only used for top-level specifications, it does not matter.

10 Case Study: Stacks with Helping

In this section we describe the how to implement and specify a concurrent stack with *helping* (also known as *cooperation*). This is an extended case study, and thus we do not present proofs in detail, but only outline the arguments and describe the important points, trusting the reader to fill in the details. We intend that the data structure should be useful in a concurrent setting, and allow threads to pass data between them: some threads may add elements and other threads might remove elements. Therefore, the data structure does not appear as a stack, in the sense of obeying the first-in last-out discipline, to any one thread. Instead, from each thread's perspective the shared data structure behaves as an unstructured bag. From a global perspective, the data structure behaves as a stack and, mxoreover, if it is only used by one thread it behaves as a stack. However the specification we give in this section is only that of a bag, with elements satisfying a given predicate. In Section 13 we will show how to give it a stronger specification, which allows us to keep more precise track of the elements being pushed and popped.

Implementation The abstract data type that we are implementing is that of a stack. Therefore, it will have two operations, push and pop. The main complication of our data structure is that push and pop must be thread-safe. One way to achieve this would be to use a lock to guard access to the stack, but this is too coarse-grained and slow when many threads wish to modify the stack concurrently.

Instead, we use a fine-grained implementation of the stack which optimistically attempts to operate on the stack without locking and then uses the compare and set primitive to check whether another thread interfered – if another thread interfered, the operation is restarted. If many threads operate on the stack concurrently it is quite likely that some of them will try to push, and some of them try to pop elements. In such a situation we can omit actually adding the element to a stack and instantly removing it. We can simply pass it from one thread to another.

This is achieved by introducing a *side-channel* for threads to communicate along. Then, if a thread attempts to get an element from a stack, it will first check whether the side-channel contains an element (which will be called an *offer*). If so, it will take the offer, but if not, the thread which wishes to get an element will instead try to get an element from the actual stack. A thread wishing to push an element will act dually; it will offer its value on the side-channel temporarily in an attempt to avoid having to compete for the main stack. The idea is that this scheme reduces contention on the main atomic cell and thus improves performance. Note that this means that a push operation *helps* a pop operation to complete (and vice versa); hence we refer to this style of implementation as using *helping* or *cooperation*.[6]

10.1 Mailboxes for Offers

As described above we will use a side-channel in the stack implementation. This side-channel can be implemented and specified separately, and this is what we do now. A side-channel has

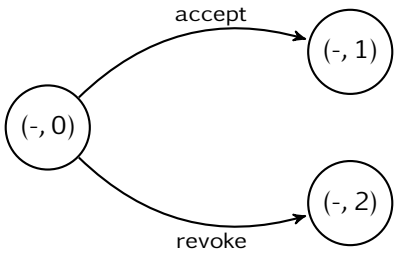
[6] In practise, the implementation of side-channels and helping is more advanced, but to illustrate the challenge of verifying implementations with helping, this simple form of helping suffices.

the following operations:

1. An offer can be *created* with an initial value.
2. An offer can be *accepted*, marking the offer as taken and returning the underlying value.
3. Once created, an offer can be *revoked* which will prevent anyone from accepting the offer and return the underlying value to the thread.

Of course, all of these operations have to be thread-safe. That is, it must be safe for an offer to be attempted to be accepted by multiple threads at once, an offer needs to be able to be revoked while it is being accepted, and so on. We choose to represent an offer as a tuple of the actual value the offer contains and a reference to an int. The underlying integer may take one of 3 values, either 0, 1 or 2. An offer of the form (v, ℓ) with $\ell \hookrightarrow 0$ is the initial state of an offer, where no one has attempted to take it, nor has it been revoked. Someone may attempt to take the offer in which case they will use a cas to switch ℓ from 0 to 1, leading to the accepted state of an offer, which is (v, ℓ) so that $\ell \hookrightarrow 1$. Revoking is almost identical but instead of switching from 0 to 1, we switch to 2.

Since both revoking and accepting an offer demand the offer to be in the initial state it is impossible for anything other than exactly one accept or one revoke to succeed. Thus the state transition system illustrating the above protocol is as follows.



The code of the three methods is quite simple, following the description above.

$$
\begin{aligned}
\text{mk\_offer} &\triangleq \lambda v.(v, \mathsf{ref}(0)) \\
\text{revoke\_offer} &\triangleq \lambda v.\ \mathsf{let}\, u = \pi_1\, v\, \mathsf{in} \\
&\qquad\qquad \mathsf{let}\, s = \pi_2\, v\, \mathsf{in} \\
&\qquad\qquad \mathsf{if}\, \mathsf{cas}(s, 0, 2)\, \mathsf{then}\, \text{Some}\, u\, \mathsf{else}\, \text{None} \\
\text{accept\_offer} &\triangleq \ \mathsf{let}\, u = \pi_1\, v\, \mathsf{in} \\
&\qquad\qquad \mathsf{let}\, s = \pi_2\, v\, \mathsf{in} \\
&\qquad\qquad \mathsf{if}\, \mathsf{cas}(s, 0, 1)\, \mathsf{then}\, \text{Some}\, u\, \mathsf{else}\, \text{None}
\end{aligned}
$$

The pattern of offering something, immediately revoking it, and returning the value if the revoke was successful is sufficiently common that we can encapsulate it in an abstraction called a *mailbox*. The idea is that a mailbox is built around an underlying cell containing an offer and that it provides two functions which, respectively, briefly put a new offer out and check for such an offer. The code for this is shown below. Note a small difference in the style from the three methods above. When the mailbox is created it does not return the underlying store, but rather returns two closures which manipulate it. This simplifies the process of using a

mailbox for stacks where we only have one mailbox at a time, but is otherwise not an important difference.

$$\begin{array}{l}\text{mailbox} \triangleq \lambda\_.\ \mathsf{let}\, r = \mathsf{ref}(\text{None})\, \mathsf{in} \\ \qquad \left(\left(\begin{array}{ll} \lambda v. & \mathsf{let}\ \text{off} = \text{mk\_offer}\, v\ \mathsf{in} \\ & r \leftarrow \text{Some off}; \text{revoke\_offer off} \end{array}\right), \left(\begin{array}{ll} \lambda\_. & \mathsf{let}\ \text{offopt} = !\, r\ \mathsf{in} \\ & \mathsf{match}\ \text{offopt}\ \mathsf{with} \\ & \quad \text{None} \Rightarrow \text{None} \\ & \mid \text{Some}\, x \Rightarrow \text{accept\_offer}\, x \\ & \mathsf{end} \end{array}\right)\right)\end{array}$$

We will call the first part of the tuple the put method, and the second one the get method. Note that in a real implementation we could, depending on contention, insert a small delay between making an offer and revoking it in the put method so that other threads would have more of a chance to accept it. Observe that the put method will return None if another thread has accepted an offer, and Some v otherwise.

10.2 The Implementation of the Stack

With an implementation of offers ready we can write the methods of the concurrent stack. As described above, we will use the mailbox as the side-channel for offers. The pop method will check whether the side-channel contains an offer using the get method, and the push method will make a temporary offer using the put method, and check the resulting value for whether the offer was accepted or not. The code for the stack is as follows. Note that this too is written in a similar style to that of mailboxes, a make function which returns two closures for the

operations rather than having them separately accept a stack as an argument.

```
stack ≜ λ_.
    let mb = mailbox() in
    let put = π1 mb in
    let get = π2 mb in
    let r = ref(None) in
    (rec pop() = match get() with
                   None    ⇒ match !r with
                               None      ⇒ None
                             | Some hd ⇒ let h = π1 hd in
                                         let t = π2 hd in
                                         if cas(r, Some hd, t)
                                          then Some h
                                          else pop()
                             end
                 | Some x ⇒ Some x
                 end,
     rec push() = match put() with
                    None    ⇒ ()
                  | Some n ⇒ let r' = !r in
                             let r'' = Some(n, r') in
                             if cas(r, r', r'') then ()
                              else push()
                  end)
```

10.3 A Bag Specification

The specification of the concurrent stack only specifies the stack's behavior as a *bag*, for reasons we described above. In particular the order of insertions is not reflected in the specification. The specification of the stack will be quite similar to the bag specification from Example 7.38, and thus the it will be parametrized by an arbitrary predicate Φ. Note that since we wrote our stack using higher-order functions, the specification of the stack method will involve nested Hoare triples, as we have seen in Section 4.2.

$$\forall \Phi.\{\mathsf{True}\}\ \mathsf{stack}()\left\{p.\exists\, \mathsf{pop}\, \mathsf{push}.\ \begin{array}{l} p = (\mathsf{pop}, \mathsf{push}) * \\ \{\mathsf{True}\}\ \mathsf{pop}()\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}\, v' * \Phi(v')\} * \\ \forall v.\, \{\Phi(v)\}\ \mathsf{push}\, v\ \{u.u = () * \mathsf{True}\} \end{array}\right\}$$

Rather than directly verifying this specification, the proof depends on several helpful lemmas verifying the behavior of offers and mailboxes. By proving these simple sublemmas, the verification of concurrent stacks can respect the abstraction boundaries constructed by isolating mailboxes as we have done.

10.4 Verifying Offers

The heart of verifying offers is accurately encoding the transition system described in the previous section. Encoding this is quite similar to the encodings of transitions systems we have seen before in Section 7.

Specifically, offers will be governed by a proposition stages which encodes what state of the three possibilities an offer is in. Ghost state is needed to ensure that certain transitions are only possible for threads with *ownership* of the offer. To this end we use the exclusive resource algebra (Example 7.18) on the singleton set. We write ex(()) for the only valid element of this resource algebra. This element will act as a token giving the owner the right to transition from the original state to the revoked state. The proposition encoding the transition system is

$$\mathsf{stages}_\gamma(v,\ell) \triangleq (\Phi(v) * \ell \hookrightarrow 0) \vee \ell \hookrightarrow 1 \vee (\ell \hookrightarrow 2 * \boxed{\mathsf{ex}(())}^\gamma)$$

Having defined this, the representation predicate is_offer is now within reach. An offer is a pair of a location containing an integer, and a value, which is the value being offered. Since multiple threads will share access to the offer, we use an invariant.

$$\mathsf{is\_offer}_\gamma(v) \triangleq \exists v', \ell.\, v = (v', \ell) * \exists \iota. \boxed{\mathsf{stages}_\gamma(v', \ell)}^\iota$$

Notice that both of these propositions are parameterized by a ghost name, γ. Each γ should uniquely correspond to an offer and represents the ownership the creator of an offer has over it, namely the right to revoke it. This is expressed in the specification of mk_offer:

$$\forall v.\, \{\Phi(v)\}\ \mathsf{mk\_offer}(v) \left\{u.\ \exists \gamma. \boxed{\mathsf{ex}(())}^\gamma * \mathsf{is\_offer}_\gamma(u)\right\}$$

This reads as that calling mk_offer will allocate an offer *as well as* returning $\boxed{\mathsf{ex}(())}^\gamma$ which represents the right to revoke an offer. The fact that ex(()) represents the right to revoke an offer can be seen in the specification for revoke_offer:

$$\forall \gamma, v. \left\{\mathsf{is\_offer}_\gamma(v) * \boxed{\mathsf{ex}(())}^\gamma\right\} \mathsf{revoke\_offer}(v) \{u.u = \mathrm{None} \vee \exists v'.\, u = \mathrm{Some}(v') * \Phi(v')\}$$

The specification for accept_offer is similar except that it does not require ownership of $\boxed{\mathsf{ex}(())}^\gamma$. This is because multiple threads may call `accept_offer` even though it will only successfully return once.

$$\forall \gamma, v. \left\{\mathsf{is\_offer}_\gamma(v)\right\} \mathsf{accept\_offer}(v) \{u.u = \mathrm{None} \vee \exists v'.\, u = \mathrm{Some}(v') * \Phi(v')\}$$

Proofs of these specifications are entirely straightforward based on what we have seen up until now, so we leave them to the reader.

10.5 Verifying Mailboxes

Having the specifications of offers in hand we can use them to give and prove specifications of the mailboxes. Since mailbox creation returns a pair of closures, specification of mailboxes will involve nested Hoare triples.

$$\{\mathsf{True}\}\ \mathsf{mailbox}() \left\{ u. \begin{array}{l} \exists\, \mathsf{put}\, \mathsf{get}. \\ u = (\mathsf{put}, \mathsf{get}) * \\ \forall v. \{\Phi(v)\}\ \mathsf{put}(v) \{w. w = \mathrm{None} \vee \exists v'.\, w = \mathrm{Some}(v') * \Phi(v')\} * \\ \{\mathsf{True}\}\ \mathsf{get}() \{w. w = \mathrm{None} \vee \exists v'.\, w = \mathrm{Some}(v') * \Phi(v')\} \end{array} \right\} \tag{34}$$

Note that the proof of this specification is made with no reference to the underlying implementation of offers, only to the specification described above. Throughout the proof an invariant is maintained governing the shared mutable cell that contains potential offers. This invariant enforces that when this cell is full, it contains an offer. It looks like this

$$\mathsf{is\_mailbox}(\ell) \triangleq \ell \hookrightarrow \mathrm{None} \vee \exists v'\gamma.\, \ell \hookrightarrow \mathrm{Some}(v') * \mathsf{is\_offer}_\gamma(v')$$

This captures the informal notion described above: either the mailbox is empty, or it contains an offer. As above, we do not show a proof of the specification and leave it to the reader.

10.6 Verifying Stacks

We now turn to the verification of stacks themselves. We have already given the desired specification above, but we repeat it here for the convenience of the reader.

$$\forall \Phi.\{\mathsf{True}\}\ \mathsf{stack}()\left\{p.\exists\, \mathrm{pop}\ \mathrm{push}.\ \begin{array}{l} p = (\mathrm{pop}, \mathrm{push}) * \\ \{\mathsf{True}\}\ \mathrm{pop}()\{v.v = \mathrm{None} \vee \exists v'.\, v = \mathrm{Some}\, v' * \Phi(v')\} * \\ \forall v.\, \{\Phi(v)\}\ \mathrm{push}\, v\, \{u.u = () * \mathsf{True}\} \end{array}\right\} \tag{35}$$

Having verified mailboxes already only a small amount of additional preparation is needed before we can prove this specification. Specifically, we need an invariant governing the shared memory cell containing the stack. The predicate $\mathsf{is\_stack}(v)$ used to form the invariant is defined as by guarded recursion as the unique predicate satisfying

$$\mathsf{is\_stack}(v) \triangleq v = \mathrm{None} \vee \exists h, t.\, v = \mathrm{Some}(h, t) * \Phi(h) * \triangleright\, \mathsf{is\_stack}(t)$$

It states that all elements of the given list satisfy the predicate Φ. Having defined this, it is straightforward to define an assertion enforcing that a location points to a stack.

$$\mathsf{stack\_inv}(v) \triangleq \exists v'.\, \ell \hookrightarrow v' * \mathsf{is\_stack}(v')$$

We will allocate an invariant containing this assertion during the proof of the stack specification. With this we can now turn to proving the specification.

To start the proof we use the Ht-let rule several times, and then the memory allocation rule, together with the specification of mailboxes, and thus we end up having to show

$$\{r \hookrightarrow \mathrm{None}\}\,(\mathrm{pop}, \mathrm{push})\left\{p.\exists\, \mathrm{pop}\ \mathrm{push}.\ \begin{array}{l} p = (\mathrm{pop}, \mathrm{push}) * \\ \{\mathsf{True}\}\ \mathrm{pop}()\{v.v = \mathrm{None} \vee \exists v'.\, v = \mathrm{Some}\, v' * \Phi(v')\} * \\ \forall v.\, \{\Phi(v)\}\ \mathrm{push}\, v\, \{u.u = () * \mathsf{True}\} \end{array}\right\}$$

where (pop, push) are the two methods in the body of the stack method. We should show this in a context where we have

$$\begin{array}{l} \forall v.\, \{\Phi(v)\}\ \mathrm{put}(v)\, \{w.w = \mathrm{None} \vee \exists v'.\, w = \mathrm{Some}(v') * \Phi(v')\} \\ \{\mathsf{True}\}\ \mathrm{get}()\, \{w.w = \mathrm{None} \vee \exists v'.\, w = \mathrm{Some}(v') * \Phi(v')\}, \end{array}$$

the specification of the mailbox. Before verifying the specifications we allocate an invariant containing stack_inv which we can, since $r \hookrightarrow \mathrm{None}$ implies stack_inv. Having done this let us verify the first method, and leave the second one for the reader. That is, let us show

$$\{\mathsf{True}\}\ \mathrm{pop}()\, \{v.v = \mathrm{None} \vee \exists v'.\, v = \mathrm{Some}\, v' * \Phi(v')\},$$

where, of course, pop is the first method in the pair returned by stack. Recall that we are verifying this a context where we have stack_inv and the above two specifications of put and get methods.

Since we are proving the correctness of a recursive function, we proceed by Löb induction, assuming

$$\triangleright \{\mathsf{True}\}\ \mathsf{pop}()\ \{v.v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}\, v' * \Phi(v')\}.$$

Using the specification of the get method we consider two cases. If the result of get() is Some x we are done, since the specification of get gives us precisely what we need. This corresponds to the fact that if an offer was made on the side-channel, then we can simply take it and we are done. Otherwise the result of get() is None and we need to use the invariant to continue with the proof. We thus open the invariant stack_inv to read $!r$, after which, by using the definition of the stack_inv predicate, we have to consider two cases. If the stack is empty ($r \hookrightarrow$ None) we are done, since the stack was empty, and thus we return None, indicating that. Otherwise we know r pointed to a pair, where Φ holds for h and t satisfies $\triangleright$ stack_inv(t). To proceed we need to open the invariant again, and after that we again consider two cases.

- If now $r \hookrightarrow$ None, then cas fails and we simply use the Löb induction hypothesis to proceed.
- Otherwise, $r \hookrightarrow \mathsf{Some}(h', t')$, where $\Phi(h')$ and $\triangleright$ stack_inv(t') hold. If the pair (h', t') is equal to (h, t) then cas succeeds and we are done, since $\Phi(h)$ holds and we can close the invariant since stack_inv(t) holds. Otherwise cas fails and we are again done by the Löb induction hypothesis.

11 The Essence of Iris

Iris 3.0 encodings.

12 Semantics

Formal semantics of the essence of Iris.

13 Logical Atomicity

The encoding of logical atomicity.

14 Types and Abstraction: Logical Relations in Iris

The encoding of unary and binary logical relations for ML types.

References

[1] J.M.E. Hyland. The effective topos. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and The Foundations of Mathematics*, pages 165–216, Amsterdam, 1982. North-Holland. 1

[2] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016. 1

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, and Derek Dreyer Lars Birkedal. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. 2017. Submitted for publication. 1

[4] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015. 1

[5] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *ESOP*, LNCS, pages 696–723, 2017. 1

[6] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 343–356, 2013. 4