Lecture Notes on
Iris: Higher-Order Concurrent Separation Logic

Lars Birkedal and Aleš Bizjak

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

April 20, 2020

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), Amin Timany (KU Leuven), Simon Friis Vindum (Aarhus University).

Contents

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 [8, 6, 9] and a longer journal paper with more details on the semantics and the latest developments of the logic [7]. 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 [4]: "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. A word of caution to the reader. The beginning of the lecture notes, until about Section 4, is rather formal and abstract. Do not be disheartened by it. This part is needed in order to fix notation, and explain the basic structure of reasoning used in concrete examples of program verification later on.

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. For the time being we refer the reader to a research paper [7] for an extensive description of the model of Iris. 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 in the notes; 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.

Acknowledgements We thank the students of the *Program Analysis and Verification* course at Aarhus University for providing feedback on an early version of these notes. We thank Ambal Guillaume, Jonas Kastberg Hinrichsen, Marianna Rapoport, and Lily Tsai for valuable feedback.

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 (*and 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 these 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.[1]

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. This is the only synchronization primitive in the language. Other synchronization constructs, such as locks, semaphores, etc., can be defined in the language, and indeed we implement and give specifications to two different kinds of locks. The

[1] iris-project.org

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, i.e., a finite set of named threads. Note that reduction of configuations is nondeterministic: we may choose to reduce in any thread in the thread pool. This reflects that we are modelling a kind of preemptive concurrent system. Any thread can be suspended at any time while some other thread gets to run. 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 [16] 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 logic. A logic is used to state and prove properties of "things". For instance, a natural number is such a "thing", as is a list of natural numbers, a value of a programming language, etc. These things need to be written in some language. In the case of Iris the underlying language of "things" is *simple type theory* with a number of basic constants. These basic constants are given by the *signature* $\mathcal{S}$.

Syntax

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

Pure reduction

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

Per-thread one-step reduction

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

Configuration reduction

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

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

Note that this language of "things" is different from the *programming language* introduced in Section 2. In fact terms of the programming language are one of the "things" we reason about. It is regrettable 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). The signature in particular contains a list of function symbols $\mathcal{F}$ with their arities, *i.e.*, their types. For a function symbol F we write $F : \tau_1, \ldots, \tau_n \to \tau_{n+1} \in \mathcal{F}$ to mean that it can be applied to a tuple of terms of types $\tau_1, \ldots, \tau_n$, and the result is of type τ_{n+1}. An example of a function symbol is addition of integers. Its arity is $\mathbb{Z}, \mathbb{Z} \to \mathbb{Z}$ where $\mathbb{Z}$ is the type of integers.

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 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.

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

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 ($-\!\!*$), which will be explained in Section 3. Then there are the later modality $\triangleright P$, explained in Section 5, and the persistently modality $\square P$, 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.

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

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

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

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

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

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

$$\frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \land Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P \lor Q : \mathsf{Prop}} \qquad \frac{\Gamma \vdash P : \mathsf{Prop} \qquad \Gamma \vdash Q : \mathsf{Prop}}{\Gamma \vdash P * Q : \mathsf{Prop}}$$

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

Figure 2: Typing Rules for Terms of the Logic

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 of rules are the standard entailment rules of intuitionistic[2] 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 12 we list additional rules which state how different connectives interact. These rules are derivable.

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.

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 $P -\!\!* Q$ 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

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

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,

[2]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.

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

$$\textsc{*-weak}\quad \frac{}{P_1 * P_2 \vdash P_1}$$

states that we can forget about resources. This makes Iris an *affine* separation logic.[3] 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

$$\textsc{*-assoc}\quad \frac{}{P_1 * (P_2 * P_3) \dashv\vdash (P_1 * P_2) * P_3} \qquad \textsc{*-comm}\quad \frac{}{P_1 * P_2 \dashv\vdash P_2 * P_1}$$

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.

[3]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.

Separating conjunction introduction The introduction rule

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

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

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

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.

Example derivation of a derivable rule We show one derivation of a derivable rule here to illustrate how the rules for separating conjunction and magic wand are used, and how these connectives interact. We leave the others as exercise for the reader.

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.[4]

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

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

[4]Indeed, for those familiar with adjoint functors, it is a consequence of the general categorical fact that left adjoints preserve colimits.

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

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

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

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

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

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

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

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

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

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

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

Laws of (affine) bunched implications.

$$\overset{*\textsc{-weak}}{\dfrac{}{P_1 * P_2 \vdash P_1}} \qquad \overset{*\textsc{-assoc}}{\dfrac{}{P_1 * (P_2 * P_3) \dashv\vdash (P_1 * P_2) * P_3}} \qquad \overset{*\textsc{-comm}}{\dfrac{}{P_1 * P_2 \dashv\vdash P_2 * P_1}} \qquad \overset{*\text{I}}{\dfrac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2}}$$

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

Figure 3: Logical entailment

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

Figure 4: Derivable rules for interaction of connectives.

Notice we made essential use of the following two rules

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

The first is the introduction rule for $\mathrel{-\!\!*}$, and the second one is easily derivable. We make use of the $\mathrel{-\!\!*}$ 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:

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

◇

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 these notes, 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 satisfying 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.

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

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

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 postconditions we use $v.Q$ 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

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

expresses that if an expression e satisfies a Hoare triple, then it will preserve any resources described by a “frame” (*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. The frame rule is essential for giving “small footprint” specifications of programs. We need only mention those resources in the pre- and post-conditions which are needed and affected by the program.

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

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

is not sound.

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

Exercise 4.2. Prove

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

False precondition The following rule

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

is trivially sound since there are no resources satisfying False.

Value and evaluation context rules

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

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 exclusive resources, exclusive in the sense that they cannot be shared. 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 any *persistent propositions* P and *any proposition* Q.

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

Intuitively, if P is persistent, then it does not depend on any exclusive resources. Thus if $P \wedge Q$ holds, then $r \in P \wedge Q$ is shareable, thus can be shared between P and Q, *i.e.*, r in $P * Q$. Later on this intuition will be slightly refined and we will see exactly what shareable means in Iris.

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 $P * Q \vdash P \wedge Q$ for any propositions P and Q. ◊

The rule of consequence

Ht-csq

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

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

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

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

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. [5] ◊

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.

Ht-load

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

[5]Note that each of those two rules is two ordinary rules, one where the top is the premise and the bottom the conclusion, and one where the bottom is the premise and top the conclusion.

Structural rules.

Ht-frame
$$\begin{array}{c} S \vdash \{P\}\, e\, \{v.Q\} \\ \hline S \vdash \{P * R\}\, e\, \{v.Q * R\} \end{array}$$

Ht-False
$$\begin{array}{c} \\ \hline S \vdash \{\mathsf{False}\}\, e\, \{v.Q\} \end{array}$$

Ht-ret
$$\begin{array}{c} w \text{ is a value} \\ \hline S \vdash \{\mathsf{True}\}\, w\, \{v.v = w\} \end{array}$$

Ht-bind
$$\begin{array}{c} E \text{ is an eval. context} \qquad S \vdash \{P\}\, e\, \{v.\, Q\} \qquad S \vdash \forall v.\, \{Q\}\, E[v]\, \{w.\, R\} \\ \hline S \vdash \{P\}\, E[e]\, \{w.\, R\} \end{array}$$

Ht-csq
$$\begin{array}{c} S \text{ persistent} \qquad S \vdash P \Rightarrow P' \qquad S \vdash \{P'\}\, e\, \{v.\, Q'\} \qquad S \vdash \forall u.\, Q'[u/v] \Rightarrow Q[u/v] \\ \hline S \vdash \{P\}\, e\, \{v.\, Q\} \end{array}$$

Ht-disj
$$\begin{array}{c} S \vdash \{P\}\, e\, \{v.\, R\} \qquad S \vdash \{Q\}\, e\, \{v.\, R\} \\ \hline\hline S \vdash \{P \vee Q\}\, e\, \{v.\, R\} \end{array}$$

Ht-exist
$$\begin{array}{c} x \notin Q \qquad S \vdash \forall x.\, \{P\}\, e\, \{v.\, Q\} \\ \hline\hline x \notin Q \qquad S \vdash \{\exists x.\, P\}\, e\, \{v.\, Q\} \end{array}$$

Rules for basic constructs of the language.

Ht-op
$$\begin{array}{c} v'' = v \circledcirc v' \\ \hline \{\mathsf{True}\}\, v \circledcirc v'\, \{r.r = v''\} \end{array}$$

Ht-load
$$\begin{array}{c} \\ \hline S \vdash \{\ell \hookrightarrow u\}\, !\,\ell\, \{v.v = u \wedge \ell \hookrightarrow u\} \end{array}$$

Ht-alloc
$$\begin{array}{c} \\ \hline S \vdash \{\mathsf{True}\}\, \mathsf{ref}(u)\, \{v.\exists \ell.\, v = \ell \wedge \ell \hookrightarrow u\} \end{array}$$

Ht-store
$$\begin{array}{c} \\ \hline S \vdash \{\ell \hookrightarrow -\}\, \ell \leftarrow w\, \{v.v = () \wedge \ell \hookrightarrow w\} \end{array}$$

Ht-rec
$$\begin{array}{c} \Gamma, g : \mathit{Val} \mid S \wedge \forall y.\, \forall v.\, \{P\}\, g v\, \{u.Q\} \vdash \forall y.\, \forall v.\, \{P\}\, e[g/f, v/x]\, \{u.Q\} \\ \hline \Gamma \mid S \vdash \forall y.\, \forall v.\, \{P\}\, (\mathsf{rec}\, f(x) = e) v\, \{u.Q\} \end{array}$$

Ht-Proj
$$\begin{array}{c} \\ \hline S \vdash \{\mathsf{True}\}\, \pi_i\, (v_1, v_2)\, \{v.v = v_i\} \end{array}$$

Ht-Match
$$\begin{array}{c} S \vdash \{P\}\, e_i\, [u/x_i]\, \{v.Q\} \\ \hline S \vdash \{P\}\, \mathsf{match}\, \mathsf{inj}_i\, u\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end}\, \{v.Q\} \end{array}$$

Ht-If
$$\begin{array}{c} \{P * v = \mathsf{true}\}\, e_2\, \{u.Q\} \qquad \{P * v = \mathsf{false}\}\, e_3\, \{u.Q\} \\ \hline \{P\}\, \mathsf{if}\, v\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3\, \{u.Q\} \end{array}$$

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

Ht-Eq
$$\begin{array}{c} S \wedge t =_\tau t' \vdash \{P\}\, e\, \{v.Q\} \\ \hline\hline S \vdash \{P \wedge t =_\tau t'\}\, e\, \{v.Q\} \end{array}$$

Ht-Ht
$$\begin{array}{c} S \wedge \{P_1\}\, e_1\, \{v.Q_1\} \vdash \{P_2\}\, e_2\, \{v.Q_2\} \\ \hline\hline S \vdash \{P_2 \wedge \{P_1\}\, e_1\, \{v.Q_1\}\}\, e_2\, \{v.Q_2\} \end{array}$$

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.

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

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

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

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.

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

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

$$\textsc{Proj}\quad \frac{}{S \vdash \{\mathsf{True}\}\,\pi_i\,(v_1, v_2)\,\{v.v = v_i\}} \qquad \textsc{Match}\quad \frac{S \vdash \{P\}\,e_i\,[u/x_i]\,\{v.Q\}}{S \vdash \{P\}\,\mathsf{match}\,\mathsf{inj}_i\,u\,\mathsf{with}\,x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2\,\mathsf{end}\,\{v.Q\}}$$

are straightforward from the operational semantics of the language.

Finally, the recursion rule is interesting.

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

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

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

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 $n \geq 0 * (n = 0) =_{Val}$ 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

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

$\diamond$

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

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

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.

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

Define the sequencing expression $e_1; e_2$ such that when this expression is run first e_1 is evaluated to a value, the value is discarded, and then e_2 is evaluated. Show the following specifications for the defined construct.

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

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

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

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

$\diamond$

Exercise 4.14. Derive the following rule.

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

$\diamond$

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

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

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.

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

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

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

Notice that while our data representation in itself does not ensure the absence of, *e.g.*, cyclic lists, the $\mathrm{isList}\,l\,xs$ 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:

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

We wish to give it the following specification:

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

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

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

and in the second

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

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)

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

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

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

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

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

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

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

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

Ht-Rec-multi

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

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.

```
rec append(l, l') = match l with
                      inj₁ x₁ ⇒ l'
                    | inj₂ x₂ ⇒ let p = ! x₂ in
                                let r = append (π₂ p) l' in
                                x₂ ← (π₁ p, r);
                                inj₂ x₂
                    end
```

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

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

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

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

  Hint: Think about what is the result of 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.

```
append' l l' = let go = rec f (h p) = match h with
                                         inj₁ x₁ ⇒ p ← (π₁ (! p), l')
                                       | inj₂ x₂ ⇒ f (π₂ (! x₂)) x₂
                                       end

               in match l with
                    inj₁ x₁ ⇒ l'
                  | inj₂ x₂ ⇒ go l x₂;
                               l
                  end
```

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 correct 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:

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

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:

```
rec rev($l, acc$) = match $l$ with
                      inj$_1$ $x_1$ ⇒ $acc$
                    | inj$_2$ $x_2$ ⇒ let $h = \pi_1\, !x_2$ in
                                 let $t = \pi_2\, !x_2$ in
                                 $x_2 \leftarrow (h, acc)$;
                                 rev($t, l$)
                    end
```

Intuitively we wish to give it the following specification:

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

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:

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

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

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

Convince yourself that $\forall xs.\, \text{reverse'}\, xs[\,] \equiv \text{reverse}\, xs$ 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

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

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 $\mathrm{reverse}[\,] +\!\!+\, us \equiv us$.

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

$$\{\mathrm{isList}\, l\ (v :: vs') * \mathrm{isList}\, acc\ us\}$$

$$l$$

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

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

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

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

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

implies the proposition

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

which is simply

$$\mathrm{isList}\, t\ vs' * \mathrm{isList}\, l\ (v :: us)$$

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

$$\begin{array}{l}
\{\mathrm{isList}\, t\ vs' * \mathrm{isList}\, l\ (v :: us)\} \\
\quad \mathsf{rev}(t, l) \\
\{r.\, \mathrm{isList}\, r(\mathrm{reverse}\, vs' +\!\!+\, v :: vs)\}
\end{array}$$

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

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

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

and prove the following three specifications

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

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.

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

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:

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

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 $v.\mathsf{inc}$ in place of $\pi_1\,v$ and analogously for $v.\mathsf{read}$.

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

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 that the counter method satisfies the above specification. ◊

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

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

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.

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

Here 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.

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

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.

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

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

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

$$\begin{aligned}\text{PrimeLoc}([]) &\equiv \mathsf{True}\\ \text{PrimeLoc}(\Phi :: \Phi s) &\equiv (\Phi(v) \twoheadrightarrow \exists n.\, v \hookrightarrow n * \text{isPrime}(n)) * \text{PrimeLoc}(\Phi s)\end{aligned}$$

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

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

The idea is that $\text{isStack}(s, xs, \Phi)$ 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 $\text{isStack}_g : Val \to \text{list}(Val \to \mathsf{Prop}) \to \mathsf{Prop}$ be the predicate associated with the general stack specificaton (7). We define $\text{isStack}_u : Val \to \text{list}\, Val \to (Val \to \mathsf{Prop}) \to \mathsf{Prop}$ 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

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

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.

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

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.

$$\forall P, I. \forall f \in Val. \forall xs. \forall l. \left\{ \begin{array}{l} \text{isList}\, l\, xs * \text{all}\, P\, xs * I\, [\,]\, a\, * \\ (\forall x \in Val. \forall a' \in Val. \forall ys. \{P x * I\, ys\, a'\}\, f\, (x, a')\, \{r. I (x :: ys) r\}) \end{array} \right\}$$
$$\mathsf{foldr}(f, a, l)$$
$$\{r. \text{isList}\, l\; xs * I\, xs\, r\}$$

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 a linked list implementing the mathematical sequence xs. This it captured by 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 $\text{all}\, P\, xs$ 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 has 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

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

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

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

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

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

and in the second

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

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

In the first case we have $\text{isList}\, l\, [\,] \equiv l = \mathsf{inj}_1()$ 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 $\text{isList}\, l\, (x :: xs') \equiv \exists hd, l'. l = \mathsf{inj}_2 hd * hd \hookrightarrow (x, l') * \text{isList}\, l'\, xs'$. By exercise 4.21 we get that the second case is taken, thus we need to prove

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

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

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

By Ht-bind-det we have to show:

1. $f\, (x, -)$ is an evaluation context.
2. the specification

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

3. the specification

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

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.[6]

[6]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.

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

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

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.

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

The specification of the sumList function is as follows.

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

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.

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

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

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

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

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

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

i.e., it is suffices to prove

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

```
filter(p, l) = let f = (λy,  let x = π₁ y in
                             let xs = π₂ y in
                             if p x
                             then inj₂ (ref(x, xs))
                             else xs)
               in
               foldr(f, inj₁ (), l)
```

Specification

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

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

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

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

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

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

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

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

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

Exercise 4.41. Prove the preceding three implications.
Hint: use induction for the second 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 7.9. 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 $\lambda_{\mathrm{ref,conc}}$ 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 Ht-Rec rule displayed in Figure 5 on page 16. [7]

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

But at present there is nothing in the logic which would allow us to do so. In contrast to the specifications of the linked list methods above, there is no tangible structure, such as the mathematical sequence representing a list, which could be used as a crutch to prove the specification by induction, for example.

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:

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

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. When using this rule we will often say we "proceed by Löb induction", or that we are going to use "Löb induction".

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 $\mathsf{True} \vdash \mathsf{False}$.

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.

[7]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.

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

$$\textsc{Later-conj}\ \begin{array}{c} R \vdash \triangleright(P \wedge Q) \\ \hline\hline R \vdash \triangleright P \wedge \triangleright Q \end{array} \qquad \textsc{Later-disj}\ \begin{array}{c} R \vdash \triangleright(P \vee Q) \\ \hline\hline R \vdash \triangleright P \vee \triangleright Q \end{array} \qquad \textsc{Later-all}\ \begin{array}{c} Q \vdash \triangleright \forall x .\, P \\ \hline\hline Q \vdash \forall x .\, \triangleright P \end{array} \qquad \textsc{Later-sep}\ \begin{array}{c} R \vdash \triangleright P * \triangleright Q \\ \hline\hline R \vdash \triangleright(P * Q) \end{array}$$

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:

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

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

Ht-let

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

Ht-let-det

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

Ht-seq

$$\frac{S \vdash \{P\}\, e_1\, \{\_.\triangleright Q\} \qquad S \vdash \{Q\}\, e_2\, \{u.R\}}{S \vdash \{P\}\, e_1; e_2\, \{u.R\}}$$

Ht-If

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

◊

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:

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

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 "persistently" 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. This could be addressed by having a separate category of propositions which could be moved in and out of preconditions, and this category would be closed under all connectives of higher-order logic. However it would still not address the issue of how to ensure that a proposition is persistent *inside the logic*. This latter issue is important when using the logic for more advanced examples. For instance it is crucial for defining Hoare triples from the notion of weakest precondition which we consider in Section 12, and for other more advanced uses of the logic. In this section we present a more uniform way of treating persistent propositions, which also addressed the second issue. We introduce a new modality $\square$ (pronounced "persistently").

The intuitive reading of $\square P$ 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, *i.e.*, those which have a duplicable part in P. Thus $\square P$ 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. As a concrete example of where this is not the case (this example is also related to the notion of invariants we introduce later) we can imagine a slight refinement of heaps as a notion of resource. Suppose that each heap is split into two parts (h_r, h_w), where h_r is read-only, and h_w is an ordinary heap which supports reading and writing. Composition of (h_r, h_w) and (h'_r, h'_w) is only defined when the components h_r and h'_r are the same, and moreover the composition of h_w and h'_w is defined as composition of ordinary heaps. Then the duplicable part of the element (h_r, h_w) is the element $(h_r, \emptyset)$, where $\emptyset$ is the empty heap.

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

$$\square P \dashv\vdash \square P * P.$$

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

We call a proposition P *persistent* if it satisfies $P \vdash \square P$. 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 $\square(x \hookrightarrow v) \dashv\vdash \mathsf{False}$.

Given the above intuitive reading of $\square P$, the first three axioms for $\square$ in Figure 7, persistently-mono, persistently-E and persistently-idemp, are immediate. These three rules together allow us to prove the following "introduction" rule for the persistently modality.

$$\frac{\square P \vdash Q}{\square P \vdash \square Q}\ \textsc{persistently-intro}\ .$$

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 $\square\mathsf{True} = \mathsf{True}$. The rest of the rules state that $\square$ 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.

Next comes the rule Persistently-sep and the derived rule Persistently-sep-derived. They govern the interaction between separating conjunction, conjunction, and the persistently modality.

$$\frac{S \vdash \square P \wedge Q}{S \vdash \square P * Q}\ \textsc{Persistently-sep}$$

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

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

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

which is equivalent to the entailment

$$\square(P \wedge Q) \vdash \square(P * Q).$$

To derive this entailment we first show the following:

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

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

$$P \vdash \square P \vdash \square P \wedge \square P \vdash \square P * \square P \vdash P * P$$

using the rule Persistently-sep in the third step, and $\square P \vdash P$ and monotonicity of separating conjunction in the last.

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

$$\square P \wedge \square Q \vdash \square(P \wedge Q) \vdash \square(P \wedge Q) * \square(P \wedge Q)$$

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

$$\square(P \wedge Q) * \square(P \wedge Q) \vdash \square P * \square Q.$$

Hence we have proved

$$\square P \wedge \square Q \vdash \square P * \square Q. \tag{12}$$

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

$$\square(P \wedge Q) \vdash \square\square(P \wedge Q) \vdash \square(\square P \wedge \square Q) \vdash \square(\square P * \square Q) \vdash \square(P * Q)$$

where in the last step we again use $\square P \vdash P$ and monotonicity of separating conjunction.

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

1. $\square\square P \vdash \square P$
2. $\square(P \Rightarrow Q) \vdash \square P \Rightarrow \square Q$
3. $P \Rightarrow Q \vdash P \twoheadrightarrow Q$
4. $\square(P \twoheadrightarrow Q) \vdash \square(P \Rightarrow Q)$
5. $\square(P \twoheadrightarrow Q) \vdash \square P \twoheadrightarrow \square Q$
6. $\square P \Rightarrow Q \vdash \square P \twoheadrightarrow Q$
7. if $P \vdash \square Q$ then $P \vdash \square Q * P$.
8. $(P \twoheadrightarrow (Q * \square R)) * P \vdash (P \twoheadrightarrow (Q * \square R)) * P * \square R$

The last two items are often useful. They state that we can get persistent propositions without consuming resources. ◊

$$\frac{P \vdash Q}{\square P \vdash \square Q}\ \textsc{persistently-mono} \qquad \frac{}{\square P \vdash P}\ \textsc{persistently-E} \qquad \frac{}{\square P \vdash \square\square P}\ \textsc{persistently-idemp}$$

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

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

$$\frac{S \vdash \square P \wedge Q}{S \vdash \square P * Q}\ \textsc{Persistently-sep}$$

Finally we have two kinds of primitive persistent propositions.

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

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

$$\textsc{Ht-persistently}\quad \frac{\square Q \wedge S \vdash \{P\}\, e\, \{v.\, R\}}{S \vdash \{P \wedge \square Q\}\, e\, \{v.\, R\}}$$

Figure 7: Laws for the persistently modality.

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.

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

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

◊

7 Concurrency: 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 $\mathrm{Some}\, x$ 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 $\mathrm{Some}\, x$ for some value x, *i.e.*, until the flag is set.

$$\mathsf{spawn} := \lambda f. \mathsf{let}\, c = \mathsf{ref}(\mathrm{None})\, \mathsf{in}\, \mathsf{fork}\, (c \leftarrow \mathrm{Some}(f\,()));c$$

$$\begin{aligned}\mathsf{join} := \mathsf{rec}\, f(c) = \ & \mathsf{match}\, !c\, \mathsf{with} \\ & \quad \mathrm{Some}\, x \Rightarrow x \\ & \mid \mathrm{None} \quad \Rightarrow f(c) \\ & \mathsf{end}\end{aligned}$$

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:

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

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 $e_1 \,\|\, e_2$ 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 which we derive in Section 7.8 from the primitive fork {} specification.

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

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.

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

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

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

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

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

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

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

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

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, thus duplicable, and hence sharable among several threads.

7.2 Invariants

To introduce invariants we need to add a type of invariant names $\mathsf{InvName}$ 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 51. As mentioned above we need to make sure that we do not open the same invariant more than once (see Example 7.3 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:

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

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

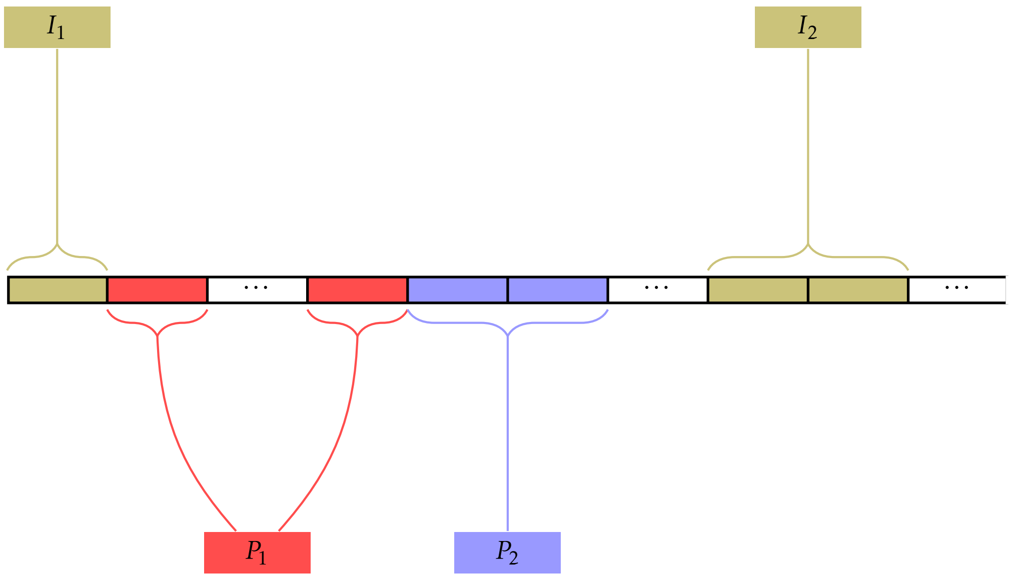
$$\textsc{Ht-inv-alloc}\quad \frac{\mathcal{E}\ \text{infinite} \qquad S \land \exists \iota \in \mathcal{E}. \boxed{P}^{\iota} \vdash \{Q\}\, e\, \{v.R\}_{\mathcal{E}}}{S \vdash \{\triangleright P * Q\}\, e\, \{v.R\}_{\mathcal{E}}}$$

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; see the explanation of the Ht-inv-open rule below.

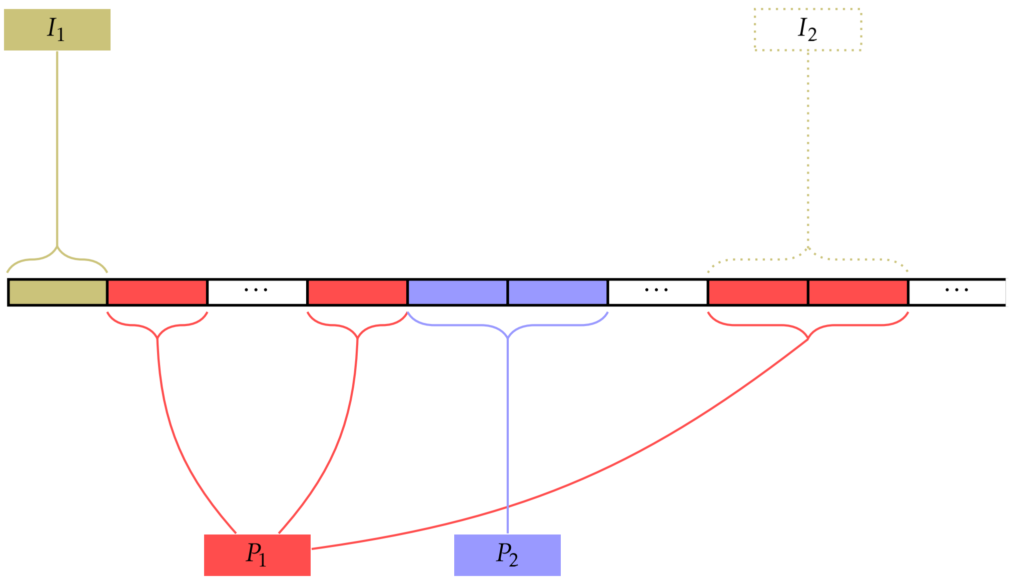
Invariants are persistent, so giving away resources to invariants is not without cost. The cost is that invariants can only be used in a restricted way, namely by the invariant opening rule.

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 7.4, 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

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

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 contains names of those invariants which we are allowed to open. We refer to $\mathcal{E}$ as a *mask*. In particular, we cannot open the same invariant twice (see Example 7.3 for an example of what goes wrong if we allow opening an invariant twice).

Note that the reader might perhaps be puzzled as to why we need the set of invariant names when we could perhaps just have the rule

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

removing the knowledge about the invariant once we open it. The reason is that this would not prevent opening invariants in a nested way because the invariant assertion $\boxed{P}^{\iota}$ is persistent. For this reason from the rule just mentioned we can easily derive

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

which would lead to a contradiction as shown in Example 7.3.

The restriction of the term e to be an *atomic* expression is also essential; see Example 7.6. 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

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

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.[8]

The stronger frame rule is the following

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

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 7.2 (Later False)**.** We will often use an invariant to tell us that some cases are impossible. For instance an invariant will often encode a transition system, e.g., encoding a communication protocol, and different threads will hold tokens which will be used to guarantee that the transition system can only be in certain states, meaning, for instance, that a certain message has not yet been sent. Thus we will hve 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.

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

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

Example 7.3 (Opening an invariant twice leads to an inconsistency)**.** This example demonstrates a problem with opening an invariant more than once. Suppose the invariant opening

[8] The precise statement and proof of this property can be found in [7], although it uses concepts we have not yet introduced here.

$$\dfrac{\textsc{Inv-persistent}}{\boxed{P}^{\iota} \vdash \Box \boxed{P}^{\iota}}$$

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

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

Figure 8: Rules for invariants.

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

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

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

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

And opening again we need to show

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

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 7.4. 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 $I = \exists m. m \geq n \land \ell \hookrightarrow m$, 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

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

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 7.5. Show this claimed specification in detail. ◊

This concludes the proof. ■

Example 7.6 (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

$$(\ell \leftarrow 4; \ell \leftarrow 3) \parallel !\ell$$

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 specification (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 analogous 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:

$$\textsc{F-duplicable}\quad \frac{}{\boxed{\mathrm{F}}^{\gamma} \vdash \boxed{\mathrm{F}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma}} \qquad \textsc{S-S-incompatible}\quad \frac{}{\boxed{\mathrm{S}}^{\gamma} * \boxed{\mathrm{S}}^{\gamma} \vdash \mathsf{False}} \qquad \textsc{S-F-incompatible}\quad \frac{}{\boxed{\mathrm{S}}^{\gamma} * \boxed{\mathrm{F}}^{\gamma} \vdash \mathsf{False}}$$

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.

$$\begin{tikzcd} \mathrm{S} \arrow[r] & \mathrm{F} \end{tikzcd}$$

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

$$\textsc{Ht-token-alloc}\quad \frac{T \in \{\mathrm{S}, \mathrm{F}\} \qquad S \vdash \left\{\exists \gamma. \boxed{T}^{\gamma} * P\right\} e \{v.Q\}}{S \vdash \{P\}\, e\, \{v.Q\}}$$

$$\textsc{Ht-token-update-pre}\quad \frac{S \vdash \left\{\boxed{\mathrm{F}}^{\gamma} * P\right\} e \{v.Q\}}{S \vdash \left\{\boxed{\mathrm{S}}^{\gamma} * P\right\} e \{v.Q\}} \qquad \textsc{Ht-token-update-post}\quad \frac{S \vdash \{P\}\, e \left\{v. \boxed{\mathrm{S}}^{\gamma} * Q\right\}}{S \vdash \{P\}\, e \left\{v. \boxed{\mathrm{F}}^{\gamma} * Q\right\}}$$

Using these rules, we now prove (16). The invariant we pick is the following predicate, parametrised by $\gamma \in$ 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 7.7. So, let us start proving. We start off by using the rule Ht-token-alloc plus Ht-exist, so we have to prove

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

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

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

$$\left\{\boxed{\mathrm{F}}^{\gamma}\right\} !\,\ell\,\{v.v \geq n+1\}. \tag{18}$$

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

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

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)

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

Using the invariant opening rule Ht-inv-open together with structural rules we need to prove

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

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

Using Ht-frame-atomic and Ht-load we get

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

Now by S-F-incompatible, the precondition is equivalent to $\triangleright$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 $\boxed{\mathrm{F}}^{\gamma}$ in the postcondition. This token 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. The notion of resources described in this section suffices for the vast majority of program verifications. However it is insufficient for certain more advanced uses of the logic, such as when Iris is used to reason about refinement, or when building the Iris program logic on top of the base logic.

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

Definition 7.8. 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 7.9. 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 are the resource algebras defined as follows.[9]

Definition 7.10 (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

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

that is, if x is valid then every sub-part of x is also valid.

The core is required to have the following properties.

$$\begin{aligned}
|a| \text{ defined} &\Rightarrow |a| \cdot a = a \\
|a| \text{ defined} &\Rightarrow ||a|| = |a| \\
a \preccurlyeq b \wedge |a| \text{ defined} &\Rightarrow |b| \text{ defined } \wedge |a| \preccurlyeq |b|.
\end{aligned}$$

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 7.11. 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 7.12. Show that in a unital resource algebra the core is always defined. Hint: $\varepsilon \preccurlyeq a$ for any element a. ◊

[9]In general, resources in Iris can be elements of so-called "cameras" which are generalizations of *resource algebras*. They add additional *approximation information* to resources which is needed for the most advanced applications, however the vast majority of program verifications needs only the "discrete cameras", which is what we call resource algebras in these lecture notes.

Example 7.13. 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 $\perp$, 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 $\perp$. Composing $\perp$ with any other element yields $\perp$. 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 $\perp$. ■

Example 7.14. 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 $X \cup \{\perp\}$, for some element $\perp$ not in X. The operation $(\cdot)$ is defined by the following rules. The non-trivial compositions are only

$$m \cdot m = m$$

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

Example 7.15 (Finite subsets of natural numbers)**.** The carrier of this resource algebra is the set of finite subsets of natural numbers and an additional element $\perp$. The operation is disjoint union, i.e.:

$$x \cdot y = \begin{cases} x \cup y & \text{if } x \cap y = \emptyset \\ \perp & \text{otherwise} \end{cases}$$

The unit of this operation is $\emptyset$. Valid elements are all finite subsets of natural numbers and the core operation maps every valid element to $\emptyset$. ■

Example 7.16. If we take the resource algebra M to have the carrier $\{\mathrm{S}, \mathrm{F}, \perp\}$ with multiplication defined as $\mathrm{F} \cdot \mathrm{F} = \mathrm{F}$ and otherwise $x \cdot y = \perp$ 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 7.17 (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. ■

Example 7.18 (Exclusive resource algebra)**.** Given a set X the exclusive resource algebra $\mathrm{Ex}(X)$ has as carrier the set X with an additional element $\perp$. The operation is defined such that $x \cdot y = \perp$ 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 $\perp$.

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 $\mathbb{N} \xrightarrow{\text{fin}} \mathrm{Ex}(\mathit{Val})$ is the same as the unital resource algebra of heaps described in Example 7.13. More precisely, show that the valid elements of $\mathbb{N} \xrightarrow{\text{fin}} \mathrm{Ex}(\mathit{Val})$ are precisely the heaps, and composition of these is exactly the same as the composition of heaps, if it is a valid element. ◊

The following examples are generic constructions. They construct resource algebras combined from a variety of smaller ones. This makes it easier to build more complex notions of resources needed in verification, since a lot of the infrastructure can be reused. However it does take some practice to get used to thinking in terms of decomposition of the desired resource algebra in terms of the smaller ones. We hope the reader will get some intuition for this by working through the example verifications in the rest of these notes.

Example 7.20 (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 $\mathcal{M}_\times$. Its carrier is the product $\mathcal{M}_1 \times \mathcal{M}_2$, and its operation is 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 7.21 (Finite map resource algebra)**.** Let $(\mathcal{M}, \mathcal{V}, |\cdot|)$ be a resource algebra. We can form a new resource algebra $\mathbb{N} \xrightharpoonup{\text{fin}} \mathcal{M}$ whose carrier is the set of partial functions from $\mathbb{N}$ 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

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

and the core is defined as

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

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

Example 7.22 (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

$$|?|_{\mathcal{M}_?} = ?$$

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

■

Extending Iris with resource algebras 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, *i.e.*, they allow us to reason about elements of the resource algebras in 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

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

Own-valid

$$\frac{}{\boxed{a : \mathcal{M}_i}^{\gamma} \vdash a \in \mathcal{V}_i}$$

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

Persistently-core

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

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.*, that all the contributions of all threads are 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 $B \subseteq \mathcal{V}$ is a *non-empty* subset of valid elements. 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 $r \in |\!\!\Rrightarrow (P * Q)$.

$$\frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash |\!\!\Rrightarrow P : \mathsf{Prop}} \qquad \overset{\textsc{upd-mono}}{\frac{P \vdash Q}{|\!\!\Rrightarrow P \vdash |\!\!\Rrightarrow Q}} \qquad \overset{\textsc{upd-intro}}{\frac{}{P \vdash |\!\!\Rrightarrow P}} \qquad \overset{\textsc{upd-idemp}}{\frac{}{|\!\!\Rrightarrow |\!\!\Rrightarrow P \vdash |\!\!\Rrightarrow P}} \qquad \overset{\textsc{upd-frame}}{\frac{}{P * |\!\!\Rrightarrow Q \vdash |\!\!\Rrightarrow (P * Q)}}$$

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.
$$\overset{\textsc{upd-sep}}{\frac{P_1 \vdash |\!\!\Rrightarrow Q_1 \qquad P_2 \vdash |\!\!\Rrightarrow Q_2}{P_1 * P_2 \vdash |\!\!\Rrightarrow (Q_1 * Q_2)}}$$

3.
$$\overset{\textsc{upd-bind}}{\frac{P_2 \vdash |\!\!\Rrightarrow Q \qquad P_1 * Q \vdash |\!\!\Rrightarrow R}{P_1 * P_2 \vdash |\!\!\Rrightarrow R}}$$

◊

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 $\mid\!\Rrightarrow Q$ from those needed to prove the $\mid\!\Rrightarrow R$. Thus we cannot use P_2 anymore when proving $\mid\!\Rrightarrow R$. Instead upd-bind very closely corresponds to the let rule in a language with an affine or linear type system.

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

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

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

$$\frac{P * P \vdash \mathsf{False}}{P \vdash \mid\!\Rrightarrow R}$$

In particular $P \vdash \mid\!\Rrightarrow \mathsf{False}$ 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.

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

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, since whether or not the program is safe to run, and its return value, only depends on the physical state.

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

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

The generalized rule of consequence is then

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

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

Exercise 7.28. Derive the following.

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

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

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

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 7.16. 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.

$$\textsc{Ht-inv-alloc-post}\quad \frac{\mathcal{E}\text{ infinite} \qquad S \vdash \{P_2\}\,e\,\{v.Q\}_{\mathcal{E}}}{S \vdash \{(\triangleright P_1) * P_2\}\,e\,\left\{v.Q \land \exists\iota \in \mathcal{E}.\boxed{P_1}^{\iota}\right\}_{\mathcal{E}}}$$

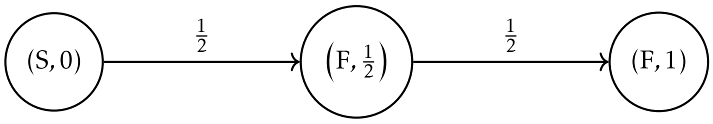
Derive the rule using Ht-inv-alloc, the fact that invariants are persistent and the generalised rule of consequence introduced above. ◊

Example 7.31. In this example we show how to use slightly more complex reasoning using resource algebras to show the specification (14) (page 47) 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 7.17, together with the resource algebra encoding the transition system with states S and F defined in Example 7.16, and used in the previous section.

The proof proceeds similarly to the proof in Example 7.7, but with a different invariant. The invariant we are going to use is

$$\begin{aligned} I(\gamma_1,\gamma_2,n) = {} & \ell \hookrightarrow n * \boxed{\mathrm{S}}^{\gamma_1} \lor \\ & \ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2} \lor \\ & \ell \hookrightarrow (n+2) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{1}^{\gamma_2} \end{aligned}$$

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


$$\begin{tikzcd} (\mathrm{S},0) \arrow[r, "\frac{1}{2}"] & (\mathrm{F},\frac{1}{2}) \arrow[r, "\frac{1}{2}"] & (\mathrm{F},1) \end{tikzcd}$$

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

$$\{\ell \hookrightarrow n\}\,(e \,\|\, e);!\ell\,\{v.v = n+1 \lor v = n+2\}.$$

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

$$\left\{\boxed{\mathrm{S}}^{\gamma_1} * \boxed{1}^{\gamma_2} * \ell \hookrightarrow n\right\}(e \,\|\, e);!\ell\,\{v.v = n+1 \lor v = n+2\}.$$

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.

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\left[1\right]^{\gamma_2}\right\}(e \parallel e); !\ell \,\{v.v = n+1 \lor v = n+2\}$$

for some ι. Using the rule Ht-seq we now verify the two parts, showing the following two triples

$$\begin{aligned}
&\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\left[1\right]^{\gamma_2}\right\}(e \parallel e)\left\{\_.\left[\mathrm{F}\right]^{\gamma_1}\right\}\\
&\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\left[\mathrm{F}\right]^{\gamma_1}\right\} !\ell \,\{v.v = n+1 \lor v = n+2\}.
\end{aligned}$$

The proof of the second triple is completely analogous to the one in Example 7.7, so we omit it here.

To show the first triple we first use the rule Own-op to get

$$\left[1\right]^{\gamma_2} \iff \left[\tfrac{1}{2}\right]^{\gamma_2} * \left[\tfrac{1}{2}\right]^{\gamma_2}$$

and

$$\left[\mathrm{F}\right]^{\gamma_1} \iff \left[\mathrm{F}\right]^{\gamma_1} * \left[\mathrm{F}\right]^{\gamma_1}$$

and hence the first triple is equivalent to

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\left[\tfrac{1}{2}\right]^{\gamma_2} * \left[\tfrac{1}{2}\right]^{\gamma_2}\right\}(e \parallel e)\left\{\_.\left[\mathrm{F}\right]^{\gamma_1} * \left[\mathrm{F}\right]^{\gamma_1}\right\}$$

which means we can use the parallel composition rule Ht-par, and we have to show

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\left[\tfrac{1}{2}\right]^{\gamma_2}\right\} e \left\{\_.\left[\mathrm{F}\right]^{\gamma_1}\right\}.$$

Recall that e is the program $\ell \leftarrow !\ell + 1$. Using the Ht-bind rule we show the following two entailments.

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \left\{\left[\tfrac{1}{2}\right]^{\gamma_2}\right\} !\ell \left\{v.(v = n \lor v = n+1) * \left[\tfrac{1}{2}\right]^{\gamma_2}\right\} \tag{19}$$

$$\boxed{I(\gamma_1,\gamma_2,n)}^{\iota} \vdash \forall v. \left\{(v = n \lor v = n+1) * \left[\tfrac{1}{2}\right]^{\gamma_2}\right\} \ell \leftarrow v+1 \left\{\_.\left[\mathrm{F}\right]^{\gamma_1}\right\} \tag{20}$$

To show (19) we open the invariant $I(\gamma_1,\gamma_2,n)$. Thus we get

$$\triangleright\left(\ell \hookrightarrow n * \left[\mathrm{S}\right]^{\gamma_1} \lor \ell \hookrightarrow (n+1) * \left[\mathrm{F}\right]^{\gamma_1} * \left[\tfrac{1}{2}\right]^{\gamma_2} \lor \ell \hookrightarrow (n+2) * \left[\mathrm{F}\right]^{\gamma_1} * \left[1\right]^{\gamma_2}\right) * \left[\tfrac{1}{2}\right]^{\gamma_2}$$

in the precondition which using the distributivity laws of the logic, together with Own-op simplifies to

$$\begin{aligned}
&\triangleright\left(\ell \hookrightarrow n * \left[\mathrm{S}\right]^{\gamma_1}\right) * \left[\tfrac{1}{2}\right]^{\gamma_2} \lor\\
&\triangleright\left(\ell \hookrightarrow (n+1) * \left[\mathrm{F}\right]^{\gamma_1} * \left[\tfrac{1}{2}\right]^{\gamma_2}\right) * \left[\tfrac{1}{2}\right]^{\gamma_2} \lor\\
&\triangleright\left(\ell \hookrightarrow (n+2) * \left[\mathrm{F}\right]^{\gamma_1} * \left[\tfrac{3}{2}\right]^{\gamma_2}\right)
\end{aligned}$$

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

$$(v = n \lor v = n+1) * \left[\tfrac{1}{2}\right]^{\gamma_2} * \triangleright I(\gamma_1,\gamma_2,n)$$

and with three preconditions corresponding to the three disjuncts above. The last is the easiest one and follows directly from rules derived in Exercise 7.2. 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.

$$\left\{\triangleright\left(\ell \hookrightarrow n * \boxed{S}^{\gamma_1}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} ! \ell \left\{v. (v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2} * \triangleright I(\gamma_1, \gamma_2, n)\right\}$$

$$\left\{\triangleright\left(\ell \hookrightarrow (n+1) * \boxed{F}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} ! \ell \left\{v. (v = n \vee v = n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2} * \triangleright I(\gamma_1, \gamma_2, n)\right\}$$

◇

Let us now turn to showing the specification (20). Using the ∀ introduction rule, distributivity of $*$ over $\vee$, and Ht-disj this means showing two specifications.

$$\boxed{I(\gamma_1, \gamma_2, n)}^{\iota} \vdash \left\{v = n * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \ell \leftarrow v + 1 \left\{\_.\boxed{F}^{\gamma_1}\right\} \tag{21}$$

$$\boxed{I(\gamma_1, \gamma_2, n)}^{\iota} \vdash \left\{v = (n+1) * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \ell \leftarrow v + 1 \left\{\_.\boxed{F}^{\gamma_1}\right\} \tag{22}$$

Let us show (21) first. Using Ht-persistently and ordinary equational reasoning the triple is equivalent to

$$\boxed{I(\gamma_1, \gamma_2, n)}^{\iota} \vdash \left\{\boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \ell \leftarrow n + 1 \left\{\_.\boxed{F}^{\gamma_1}\right\}$$

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

$$\triangleright\left(\ell \hookrightarrow n * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \vee \triangleright\left(\ell \hookrightarrow (n+1) * \boxed{F}^{\gamma_1} * \boxed{1}^{\gamma_2}\right) \vee \triangleright\left(\ell \hookrightarrow (n+2) * \boxed{F}^{\gamma_1} * \boxed{\tfrac{3}{2}}^{\gamma_2}\right)$$

in the precondition. As before the last disjunct is equivalent to $\triangleright$False so we may deal with it exactly as before using results from Exercise 7.2. Using Ht-disj we thus need to prove the following two specifications.

$$\left\{\triangleright\left(\ell \hookrightarrow n * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right)\right\} \ell \leftarrow n + 1 \left\{\_.\boxed{F}^{\gamma_1} * \triangleright I(\gamma_1, \gamma_2, n)\right\} \tag{23}$$

$$\left\{\triangleright\left(\ell \hookrightarrow (n+1) * \boxed{F}^{\gamma_1} * \boxed{1}^{\gamma_2}\right)\right\} \ell \leftarrow n + 1 \left\{\_.\boxed{F}^{\gamma_1} * \triangleright I(\gamma_1, \gamma_2, n)\right\} \tag{24}$$

To show the first specification we start by using Ht-frame-atomic to get

$$\left\{\triangleright\left(\ell \hookrightarrow n * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right)\right\} \ell \leftarrow n + 1 \left\{\ell \hookrightarrow n + 1 * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right\} \tag{25}$$

We then use the fact that $S \rightsquigarrow F$ together with the rule Ghost-update, and upd-frame to get

$$\left(\ell \hookrightarrow n + 1 * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \Rrightarrow \left(\ell \hookrightarrow n + 1 * \boxed{F}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right)$$

Moreover $F = F \cdot F$ and thus Own-op gives us $\boxed{F}^{\gamma_1} \Rrightarrow \left(\boxed{F}^{\gamma_1} * \boxed{F}^{\gamma_1}\right)$ and thus together we have

$$\left(\ell \hookrightarrow n + 1 * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \Rrightarrow \left(\ell \hookrightarrow n + 1 * \boxed{F}^{\gamma_1} * \boxed{F}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right).$$

Clearly $\ell \hookrightarrow n + 1 * \boxed{F}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2} \Rrightarrow \triangleright I(\gamma_1, \gamma_2, n)$ and thus we have

$$\left(\ell \hookrightarrow n + 1 * \boxed{S}^{\gamma_1} * \boxed{\tfrac{1}{2}}^{\gamma_2}\right) \Rrightarrow \left(\boxed{F}^{\gamma_1} * \triangleright I(\gamma_1, \gamma_2, n)\right)$$

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 $\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\frac{1}{2}}^{\gamma_2}$. Analogously, if the invariant was in state $\ell \hookrightarrow (n+1) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{\frac{1}{2}}^{\gamma_2}$ when reading then after writing it was in state $\ell \hookrightarrow (n+2) * \boxed{\mathrm{F}}^{\gamma_1} * \boxed{1}^{\gamma_2}$. Finally, we have used the fact that the thread owned $\boxed{\frac{1}{2}}^{\gamma_2}$ 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

$$\{\ell \hookrightarrow n\}\,((e \parallel e) \parallel e); !\ell\,\{v. v = n+1 \lor v = n+2 \lor v = n+3\}.$$

◊

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 indeed not operationally equivalent, since the value at ℓ can be changed by some other thread before $\ell \leftarrow v_2$ 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

$$\frac{}{\{\triangleright \ell \hookrightarrow v\}\,\mathsf{cas}(\ell, v_1, v_2)\,\{u.(u = \mathsf{true} * v = v_1 * \ell \hookrightarrow v_2) \lor (u = \mathsf{false} * v \neq v_1 * \ell \hookrightarrow v)\}}$$

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 module 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:

```
let newLock() = ref(false)
let acquire l = if cas(l, false, true) then () else acquire l
let release l = l ← false
```

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:

$$\begin{aligned}
&\exists\, \mathrm{isLock} : \mathit{Val} \to \mathsf{Prop} \to \mathsf{GhostName} \to \mathsf{Prop}.\\
&\exists\, \mathrm{locked} : \mathsf{GhostName} \to \mathsf{Prop}.\\
&\quad\ \square(\forall P, v, \gamma.\ \mathrm{isLock}(v, P, \gamma) \Rightarrow \square\mathrm{isLock}(v, P, \gamma))\\
&\wedge\quad \square(\forall \gamma.\ \mathrm{locked}(\gamma) * \mathrm{locked}(\gamma) \Rightarrow \mathsf{False})\\
&\wedge\quad \forall P. \{P\}\ \mathrm{newLock}()\ \{v.\exists \gamma.\ \mathrm{isLock}(v, P, \gamma)\}\\
&\wedge\quad \forall P, v, \gamma. \{\mathrm{isLock}(v, P, \gamma)\}\ \mathrm{acquire}\, v\ \{\_.P * \mathrm{locked}(\gamma)\}\\
&\wedge\quad \forall P, v, \gamma. \{\mathrm{isLock}(v, P, \gamma) * P * \mathrm{locked}(\gamma)\}\ \mathrm{release}\, v\ \{\_.\mathsf{True}\}
\end{aligned}$$

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 call 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 thus 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:

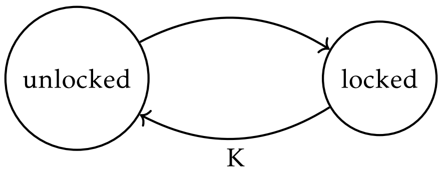
$$I(\ell, P, \gamma) = \ell \hookrightarrow \mathsf{false} * \left[\mathrm{K}\right]^{\gamma} * P \vee \ell \hookrightarrow \mathsf{true}.$$

With this we define the isLock and locked predicates as follows.

$$\begin{aligned}
\mathrm{isLock}(v, P, \gamma) &= \exists \ell \in \mathit{Loc}, \iota \in \mathsf{InvName}.\ v = \ell \wedge \boxed{I(\ell, P, \gamma)}^{\iota}\\
\mathrm{locked}(\gamma) &= \left[\mathrm{K}\right]^{\gamma}
\end{aligned}$$

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 $\text{locked}(\gamma)$, and together with the fact that $\text{locked}(\gamma)$ 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 $\text{isLock}(v,P,\gamma)$ is persistent, which it is because invariants and equality are persistent, and conjunction and existential quantification preserves persistency.

The second says that $\text{locked}(\gamma)$ *is not* duplicable. This follows as $\text{K}\cdot\text{K} = \bot$ by definition of the resource algebra: $\boxed{\text{K}}^{\gamma} * \boxed{\text{K}}^{\gamma} \vdash \boxed{\text{K}\cdot\text{K}}^{\gamma}$ 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:

$$\{P\}\,\text{newLock}()\,\{v.\exists\gamma.\ \text{isLock}(v,P,\gamma)\}$$

By Ht-beta, it suffices to show

$$\{P\}\,\text{ref}(\text{false})\,\{v.\exists\gamma.\ \text{isLock}(v,P,\gamma)\}$$

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

$$\{\text{locked}(\gamma) * P\}\,\text{ref}(\text{false})\,\{v.\exists\gamma.\ \text{isLock}(v,P,\gamma)\}$$

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

$$\forall v,P,\gamma.\,\{\triangleright\,\text{isLock}(v,P,\gamma)\}\ \text{acquire}\,v\,\{\_.P * \text{locked}(\gamma)\} \tag{26}$$

we show the following triple

$$\{\text{isLock}(v,P,\gamma)\}\,\text{if}\,\text{cas}(v,\text{false},\text{true})\,\text{then}\,()\,\text{else}\,\text{acquire}(v)\,\{\_.P * \text{locked}(\gamma)\}.$$

The isLock predicate gives us that v is a location ℓ governed by an invariant, which we can move into the context as follows:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\text{True}\}\,\text{if}\,\text{cas}(\ell,\text{false},\text{true})\,\text{then}\,()\,\text{else}\,\text{acquire}(\ell)\,\{\_.P * \text{locked}(\gamma)\}$$

We next evaluate the cas expression with the Ht-bind rule. As our intermediate step we proceed to show the following triple:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{\mathsf{True}\}\,\mathsf{cas}(\ell,\mathsf{false},\mathsf{true})\,\{u.(u=\mathsf{true} * P * \mathrm{locked}(\gamma)) \lor (u=\mathsf{false})\}.$$

As cas is atomic, we open the invariant to get at ℓ, using Ht-inv-open, and it suffices to show that

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \begin{array}{c}\{\triangleright I(\ell,P,\gamma)\}\\ \mathsf{cas}(\ell,\mathsf{false},\mathsf{true})\\ \{u.((u=\mathsf{true} * P * \mathrm{locked}(\gamma)) \lor (u=\mathsf{false})) * I(\ell,P,\gamma)\}\end{array}.$$

We proceed by cases on the invariant (using the rule Ht-disj). In the first case we need to show

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \begin{array}{c}\{\triangleright(\ell \hookrightarrow \mathsf{false} * \mathrm{locked}\,\gamma * P)\}\\ \mathsf{cas}(\ell,\mathsf{false},\mathsf{true})\\ \{u.(u=\mathsf{true} * P * \mathrm{locked}(\gamma) \lor (u=\mathsf{false})) * I(\ell,P,\gamma)\}\end{array}.$$

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

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \begin{array}{c}\{\triangleright(\ell \hookrightarrow \mathsf{true})\}\\ \mathsf{cas}(\ell,\mathsf{false},\mathsf{true})\\ \{u.((u=\mathsf{true} * P * \mathrm{locked}(\gamma) \lor (u=\mathsf{false})) * I(\ell,P,\gamma)\}\end{array}.$$

Again we strengthen the post-condition, this time to $u=\mathsf{false} * \ell \hookrightarrow \mathsf{true}$, and we are done by using the rule 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:

$$\boxed{I(\ell,P,\gamma)}^{\iota} \vdash \{u=\mathsf{true} * P * \mathrm{locked}(\gamma) \lor u=\mathsf{false}\}\,\mathsf{if}\,u\,\mathsf{then}\,()\,\mathsf{else}\,\mathrm{acquire}\,\ell\,\{\_.P * \mathrm{locked}(\gamma)\}$$

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, isLock(v, P, γ) 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:

$$\left\{\boxed{I(\ell, P, \gamma)}^{\iota} * P * \text{locked}(\gamma)\right\} \ell \leftarrow \mathsf{false} \{\_.\mathsf{True}\}$$

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:

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \{\triangleright I(\ell, P, \gamma) * P * \text{locked}(\gamma)\} \ell \leftarrow \mathsf{false} \{\_.\triangleright I(\ell, P, \gamma)\}$$

We consider two cases, based on the disjunction in $I(\ell, P, \gamma)$ in the precondition. The first case is

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow \mathsf{false} * \text{locked}(\gamma) * P) * P * \text{locked}(\gamma)\} \ell \leftarrow \mathsf{false} \{\_.\triangleright I(\ell, P, \gamma)\}$$

which is inconsistent as $\text{locked}(\gamma) * \text{locked}(\gamma) \vdash \mathsf{False}$, as argued above. We are done by Ht-later-false. In the second case we need to prove

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow \mathsf{true}) * P * \text{locked}(\gamma)\} \ell \leftarrow \mathsf{false} \{\_.\triangleright I(\ell, P, \gamma)\}$$

and in the postcondition we chose the first disjunct by Ht-csq – *i.e.*, we will show the following triple:

$$\boxed{I(\ell, P, \gamma)}^{\iota} \vdash \{\triangleright(\ell \hookrightarrow \mathsf{true}) * \triangleright(P * \text{locked}(\gamma))\} \ell \leftarrow \mathsf{false} \{\_.\triangleright(\ell \hookrightarrow \mathsf{false}) * \triangleright(\text{locked}(\gamma) * P)\}$$

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.

```
      let newBag = λ_. (ref(None), newLock())
    let insert = λx. λv. let ℓ = π₁ x in
                         let lock = π₂ x in
                         acquire lock;
                         ℓ ← Some(v, !ℓ);
                         release lock
      let remove = λx. let ℓ = π₁ x in
                         let lock = π₂ x in
                         acquire lock;
                         let r = match !ℓ with
                                     None    ⇒ None
                                   | Some p  ⇒ ℓ ← π₂ p; Some(π₁ p)
                                 end
                         in release lock; r
```

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:

$$\begin{aligned}
&\exists \mathrm{isBag} : (\mathit{Val} \to \mathsf{Prop}) \times \mathit{Val} \to \mathsf{Prop}.\\
&\forall (\Phi : \mathit{Val} \to \mathsf{Prop}).\\
&\quad \square(\forall b.\ \mathrm{isBag}(\Phi, b) \Rightarrow \square \mathrm{isBag}(\Phi, b))\\
&\wedge \quad \{\mathsf{True}\}\ \mathrm{newBag}()\ \{b.\ \mathrm{isBag}(\Phi, b)\}\\
&\wedge \quad \forall b u.\ \{\mathrm{isBag}(\Phi, b) * \Phi(u)\}\ \mathrm{insert}\ b\ u\ \{\_.\mathsf{True}\}\\
&\wedge \quad \forall b.\ \{\mathrm{isBag}(\Phi, b)\}\ \mathrm{remove}\ b\ \{v. v = \mathrm{None} \vee \exists x.\ v = \mathrm{Some}\ x \wedge \Phi(x)\}
\end{aligned}$$

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:

$$\mathrm{isBag}(\Phi, b) = \exists \ell v \gamma.\ b = (\ell, v) \wedge \mathrm{isLock}(v, \exists xs.\ \ell \hookrightarrow xs * \mathrm{bagList}(\Phi, xs), \gamma)$$

where bagList is defined by guarded recursion as the unique predicate satisfying

$$\mathrm{bagList}(\Phi, xs) = xs = \mathrm{None} \vee \exists x. \exists r.\ xs = \mathrm{Some}(x, r) \wedge \Phi(x) * \triangleright(\mathrm{bagList}(\Phi, r)).$$

Let $\Phi : \mathit{Val} \to \mathsf{Prop}$ be arbitrary.

Exercise 7.39. Prove that $\mathrm{isBag}(\Phi, b)$ is persistent for any b. ◊

Exercise 7.40. Prove the newBag specification:

$$\{\mathsf{True}\}\,\mathrm{newBag}()\,\{b.\,\mathrm{isBag}(\Phi, b)\}. \qquad ◊$$

Note that since $\mathrm{isBag}(\Phi, b)$ is persistent for any b we can derive, by using the frame rule (exercise!), the following specification

$$\forall b.\,\{\mathrm{isBag}(\Phi, b)\}\,\mathrm{remove}\,b\,\{v.\,(v = \mathrm{None} \vee \exists x.\, v = \mathrm{Some}\,x \wedge \Phi(x)) * \mathrm{isBag}(\Phi, b)\}$$

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

$$\{\mathrm{isBag}(\Phi, b)\}\,\mathrm{remove}\,b\,\{v.v = \mathrm{None} \vee \exists x.\, v = \mathrm{Some}\,x \wedge \Phi(x)\}$$

for some value b.

By definition of $\mathrm{isBag}(\Phi, b)$ and by using HT-EXIST, and then HT-PERSISTENTLY together with HT-EQ we have to prove

$$\{\mathrm{isLock}(\mathrm{lock}, \exists xs.\, \ell \hookrightarrow xs * \mathrm{bagList}(\Phi, xs), \gamma)\}\,\mathrm{remove}(\ell, \mathrm{lock})\,\{u.u = \mathrm{None} \vee \exists x.\, u = \mathrm{Some}\,x \wedge \Phi(x)\}$$

for some ℓ, lock and γ. Using HT-BETA and HT-LET-DET we reduce to showing

$$\{\mathrm{isLock}(\mathrm{lock}, \exists xs.\, \ell \hookrightarrow xs * \mathrm{bagList}(\Phi, xs), \gamma)\}\,e\,\{u.u = \mathrm{None} \vee \exists x.\, u = \mathrm{Some}\,x \wedge \Phi(x)\}$$

where e is the program

```
 acquire lock;
let r = match !ℓ with
        None    ⇒ None
      | Some p ⇒ ℓ ← π₂ p; Some(π₁ p)
      end
in release lock; r
```

Using the sequencing rule HT-SEQ we use the specification of acquire as the first triple, and thus we have to prove

$$\{\mathrm{locked}(\gamma) * \exists xs.\, \ell \hookrightarrow xs * \mathrm{bagList}(\Phi, xs)\}\,e'\,\{u.u = \mathrm{None} \vee \exists x.\, u = \mathrm{Some}\,x \wedge \Phi(x)\}$$

where e' is the part of program e after acquire.

Using the fact that $\exists$ and $\vee$ distribute over $*$ (see Figure 4 on page 12), HT-EXIST and then the definition of $\mathrm{bagList}(\Phi, xs)$ together with HT-DISJ we consider two cases. The first case is

$$\{\mathrm{locked}(\gamma) * \ell \hookrightarrow xs * xs = \mathrm{None}\}\,e'\,\{u.u = \mathrm{None} \vee \exists x.\, u = \mathrm{Some}\,x \wedge \Phi(x)\}$$

Exercise 7.41. Prove the above triple (possibly after looking at the proof of the second case below). ◊

In the second case, after using rules in Figure 4 and Ht-exist, we get the following proof obligation.

$$\{\mathrm{locked}(\gamma) * \ell \hookrightarrow xs * xs = \mathrm{Some}(x,r) * \Phi(x) * \triangleright \mathrm{bagList}(\Phi,r)\}\, e' \,\{u. u = \mathrm{None} \vee \exists x.\, u = \mathrm{Some}\, x \wedge \Phi(x)\}$$

which simplifies, by Ht-Eq and the rule of consequence, to proving

$$\{\mathrm{locked}(\gamma) * \ell \hookrightarrow \mathrm{Some}(x,r) * \Phi(x) * \triangleright \mathrm{bagList}(\Phi,r)\}\, e' \,\{u. \exists x.\, u = \mathrm{Some}\, x \wedge \Phi(x)\}.$$

We use the let rule Ht-let-det. For the first premise we show

```
{locked(γ) ∗ ℓ ↪ Some(x,r) ∗ Φ(x) ∗ ▷bagList(Φ,r)}
    match !ℓ with
      None   ⇒ None
    | Some p ⇒ ℓ ← π₂ p; Some(π₁ p)
    end
{u.u = Some x ∧ ℓ ↪ r ∗ Φ(x) ∗ locked(γ) ∗ bagList(Φ,r)}
```

(note the omission of ▷ 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

$$\{\mathrm{locked}(\gamma) * \ell \hookrightarrow \mathrm{Some}(x,r) * \Phi(x) * \triangleright \mathrm{bagList}(\Phi,r)\}$$
$$\ell \leftarrow \pi_2(x,r); \mathrm{Some}(\pi_1(x,r))$$
$$\{u. u = \mathrm{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi,r)\}$$

which by using Ht-bind and Ht-Proj simplifies to showing

$$\{\mathrm{locked}(\gamma) * \ell \hookrightarrow \mathrm{Some}(x,r) * \Phi(x) * \triangleright \mathrm{bagList}(\Phi,r)\}$$
$$\ell \leftarrow r; \mathrm{Some}(\pi_1(x,r))$$
$$\{u. u = \mathrm{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi,r)\}$$

Using the sequencing rule we first show

$$\{\mathrm{locked}(\gamma) * \ell \hookrightarrow \mathrm{Some}(x,r) * \Phi(x) * \triangleright \mathrm{bagList}(\Phi,r)\}$$
$$\ell \leftarrow r$$
$$\{\_. \ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi,r)\}$$

by using Ht-frame-atomic to remove the ▷ on bagList. The triple, the second premise of the consequence rule,

$$\{\ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi,r)\}$$
$$\mathrm{Some}(\pi_1(x,r))$$
$$\{u. u = \mathrm{Some}\, x \wedge \ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi,r)\}$$

is then easy to establish. Exercise!

Finally, we need to establish the second premise of the rule Ht-let-det, which means showing

$$\{\ell \hookrightarrow r * \Phi(x) * \mathrm{locked}(\gamma) * \mathrm{bagList}(\Phi,r)\}$$
$$\mathrm{release}\, \mathrm{lock}; \mathrm{Some}\, x$$
$$\{u. \exists x.\, u = \mathrm{Some}\, x \wedge \Phi(x)\}$$

We can use the sequencing rule together with the release specification to give away the resources $\ell \hookrightarrow r$, $\text{locked}(\gamma)$ and $\text{bagList}(\Phi, r)$ back to the lock. We are left with proving

$$\begin{aligned}&\{\Phi(x)\}\\&\quad \text{Some}\, x\\&\{u.\exists x.\, u = \text{Some}\, x \wedge \Phi(x)\}\end{aligned}$$

which is immediate.

Remark 7.42. 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. However in Iris it is possible to give a stronger specification to the lock module which can ensure that locks which are acquired must be released, however this requires a more sophisticated use of the Iris base logic, so we do not dwell on it here, and instead focus only on safety specifications. ■

Exercise 7.43. Prove the specification of the insert method:

$$\forall b u.\, \{\text{isBag}(\Phi, b) * \Phi(u)\}\, \text{insert}\, b\, u\, \{\_.\text{True}\} \qquad \diamond$$

■

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 $\text{isCounter}(v, n)$ 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 $\text{isCounter}(v, n)$ 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, and moreover the increment returns the original value of the counter, of which we know the lower bound.

$$\forall v.\forall n.\, \{\text{isCounter}(v, n)\}\, \text{incr}\, v\, \{u.u \geq n * \text{isCounter}(v, n+1)\}.$$

Following the discussion above, when reading the value of the counter it might be larger than that known to us via the isCounter predicate, *i.e.*, other threads might have increased its value without us knowing it.

$$\forall v.\forall n.\, \{\text{isCounter}(v, n)\}\, \text{read}\, v\, \{u.u \geq n\}$$

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.

$$\text{newCounter}() = \texttt{ref}(0)$$

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.

$$\begin{aligned}\texttt{rec}\,\text{incr}(\ell) = {} & \texttt{let}\, n = !\ell\, \texttt{in} \\ & \texttt{let}\, m = n + 1\, \texttt{in} \\ & \texttt{if}\, \texttt{cas}(\ell, n, m)\, \texttt{then}\, n\, \texttt{else}\, \text{incr}\, \ell\end{aligned}$$

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, $\exists m. \ell \hookrightarrow m$, 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

$$\exists m. \ell \hookrightarrow m \wedge m \geq n$$

and thus have

$$\text{isCounter}(\ell, n) = \exists \iota. \boxed{\exists m. \ell \hookrightarrow m \wedge m \geq n}^{\iota}.$$

This is an invariant, however it is not strong enough. We cannot prove the increment method using this invariant since we cannot update isCounter(ℓ, n) to 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 isCounter(ℓ, 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

$$\text{isCounter}(\ell, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists \iota. \boxed{\exists m. \ell \hookrightarrow m * \boxed{\bullet m}^{\gamma}}^{\iota} \tag{27}$$

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 isCounter$(\ell, n, \gamma)$ is to be persistent, it must be that $\boxed{\circ n}^{\gamma}$ is persistent, which is only the case (see Persistently-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 are 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.

$$|\circ n| = \circ n \tag{28}$$

$$\bullet m \cdot \circ n \in \mathcal{V} \Rightarrow m \geq n \tag{29}$$

$$\bullet m \cdot \circ n \rightsquigarrow \bullet (m+1) \cdot \circ (n+1) \tag{30}$$

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 (29) 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 (29) holds.

Requirement (28) 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 (30). Recall Definition 7.23 (on page 60) 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 7.44. Let isCounter : $Val \to \mathbb{N} \to \mathsf{GhostName} \to \mathsf{Prop}$ be the predicate

$$\text{isCounter}(\ell, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists \iota. \boxed{\exists m. \ell \hookrightarrow m * \boxed{\bullet m}^{\gamma}}^{\iota}.$$

Show the following specifications for the methods defined above.

$$\begin{array}{l}
\{\mathsf{True}\}\,\text{newCounter}()\,\{u.\exists \gamma.\ \text{isCounter}(u, 0, \gamma)\}\\
\forall \gamma. \forall v. \forall n. \{\text{isCounter}(v, n, \gamma)\}\ \text{read}\, v\, \{u.u \geq n\}\\
\forall \gamma. \forall v. \forall n. \{\text{isCounter}(v, n, \gamma)\}\ \text{incr}\, v\, \{u.u \geq n * \text{isCounter}(v, n+1, \gamma)\}
\end{array}$$
◊

Exercise 7.45. 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.

$$\{\mathsf{True}\}\, e\, \{v.v \geq 1\}.$$
◊

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 7.44. 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 whether there are other threads using the counter, and thus 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 specifications.

$$\begin{array}{l}
\forall \gamma. \forall v. \forall n. \{\text{isCounter}(v, n, \gamma, 1)\}\ \text{read}\, v\, \{u.u = n\}\\
\forall q. \forall \gamma. \forall v. \forall n. \{\text{isCounter}(v, n, \gamma, q)\}\ \text{read}\, v\, \{u.u \geq n\}
\end{array}$$

The increment has an analogous 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. We achieve this by parameterising the counter predicate with an additional fraction $p \in (0,1]$. This fraction indicates our degree of knowledge about the counter. If $p = 1$ we have the full ownership of the counter, and thus can know its exact value. Otherwise we only know its lower bound as with the previous counter specification. By using the following splitting property of the new isCounter predicate

$$\text{isCounter}(v,n,\gamma,p) * \text{isCounter}(v,m,\gamma,q) \dashv\vdash \text{isCounter}(v,n+m,\gamma,p+q)$$

we can create the counter, share it among different threads, and then once all of those threads are finished executing we can combine all the counters and read off the exact value.

In light of this rule there is another way to read the assertion isCounter(v,n,γ,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 7.46 (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 7.12)

$$|(x,a)|_{\textsc{Auth}(\mathcal{M})} = (\bot, |a|)$$

and the set of valid elements is

$$\mathcal{V}_{\textsc{Auth}(\mathcal{M})} = \left\{(x,a) \,\middle|\, x = \bot \land a \in \mathcal{V} \lor x \in \mathcal{M} \land x \in \mathcal{V} \land a \preccurlyeq x\right\}$$

We write $\bullet m$ for (m,ε) and $\circ n$ for $(\bot, n)$. ■

Exercise 7.47. Show that the resource algebra $\mathcal{M}$ we used in the counter specification in Exercise 7.44 is exactly the resource algebra $\textsc{Auth}(\mathbb{N}_{\max})$ where $\mathbb{N}_{\max}$ is the resource algebra whose carrier is the set of natural numbers, and its operation the maximum. Its core is the identity function and all elements are valid. ◊

Exercise 7.48. 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 $\textsc{Auth}(\mathcal{M})$.
- if $x \cdot z$ is valid in $\mathcal{M}$ and $w \preccurlyeq z$ then
$$\bullet x \cdot \circ y \rightsquigarrow \bullet(x \cdot z) \cdot \circ(y \cdot w)$$
in $\textsc{Auth}(\mathcal{M})$.

$\diamondsuit$

Recall the isCounter predicate we used previously

$$\text{isCounter}(\ell, n, \gamma) = \boxed{\circ n}^{\gamma} * \exists \iota. \boxed{\exists m. \ell \hookrightarrow m * \boxed{\bullet m}^{\gamma}}^{\iota}.$$

We wish to incorporate into it the fraction p indicating how much of the counter this thread owns. We do this as follows.

$$\text{isCounter}(\ell, n, \gamma, p) = \boxed{\circ(p, n)}^{\gamma} * \exists \iota. \boxed{\exists m. \ell \hookrightarrow m * \boxed{\bullet(1, m)}^{\gamma}}^{\iota}.$$

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 7.17,
- $\mathbb{N}$ 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 7.49. Show the following properties of the resource algebra $(\mathbb{Q}_{01} \times \mathbb{N})_?$.

- $(p, n) \cdot (q, m) = (p + q, n + m)$
- if $q \leq 1$ then $(p, n) \preccurlyeq (q, m)$ if and only if
  - $p \leq q$ and $n \leq m$ (where $\leq$ is the standard ordering on naturals and rationals)
  - if $p = 1$ then $q = 1$ and $n = m$.

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 $n = m$
- $\bullet(1,m)\cdot\circ(p,n) \rightsquigarrow \bullet(1,m+1)\cdot\circ(p,n+1)$. $\diamond$

Using the results from the preceding exercise about ghost updates the following exercise is a straightforward adaptation of Exercise 7.44.

Exercise 7.50. Let isCounter be the predicate

$$\text{isCounter}(\ell,n,\gamma,p) = \boxed{\circ(p,n)}^{\gamma} * \exists\iota.\,\boxed{\exists m.\,\ell \hookrightarrow m * \boxed{\bullet(1,m)}^{\gamma}}^{\iota}.$$

First show that for any $p, q, n,$ and k, 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.

$$\begin{array}{l}
\{\mathsf{True}\}\,\text{newCounter}()\,\{u.\exists\gamma.\,\text{isCounter}(u,0,\gamma,1)\}\\
\forall p.\forall\gamma.\forall v.\forall n.\,\{\text{isCounter}(v,n,\gamma,p)\}\,\text{read}\,v\,\{u.u \geq n\}\\
\forall\gamma.\forall v.\forall n.\,\{\text{isCounter}(v,n,\gamma,1)\}\,\text{read}\,v\,\{u.u = n\}\\
\forall p.\forall\gamma.\forall v.\forall n.\,\{\text{isCounter}(v,n,\gamma,p)\}\,\text{incr}\,v\,\{u.u \geq n * \text{isCounter}(v,n+1,\gamma,p)\}
\end{array}$$
$\diamond$

Using the specification from the previous exercise we can now revisit Exercise 7.45 to give it the most precise specification.

Exercise 7.51. 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.

$$\{\mathsf{True}\}\,e\,\{v.v = 2\}.$$
$\diamond$

7.8 Parallel composition via fork {}

We can now show how to derive the proof rule Ht-par which we simply stated on page 46 from the following rule for the primitive fork {}. The primitive fork {} rule is as follows.

$$\textsc{Ht-fork}\quad \frac{S \vdash \{P\}\,e\,\{\_.\mathsf{True}\}_{\mathcal{E}}}{S \vdash \{P\}\,\mathsf{fork}\,\{e\}\,\{v.v = ()\}_{\mathcal{E}}}$$

The key point is that we only require the forked-off thread to be safe, we do not care about its return value, hence the post-condition True in the premise.

Recall how parallel composition is defined in Section 7.1. It uses two auxiliary methods spawn, which creates a new thread and a “shared channel” which is used to signal when it is done, and the join method which listens on the channel until the spawned thread is done. These methods are of independent interest since, in contrast to the plain fork {}, they allow us to wait for a forked-off thread to finish. Since we are using a shared channel we will use an invariant to allow the two threads to access it.

The method spawn has a function f as an argument. Let us assume that f has the following specification for some predicates P and $v.Q$.

$$\{P\}\, f\,()\, \{v.Q\}.$$

With these parameters fixed we define the following predicate

$$\mathrm{joinHandle}(\ell, \gamma, v.Q) \triangleq \boxed{()}^{\gamma} * \boxed{\ell \hookrightarrow \mathrm{None} \vee \exists x.\, \ell \hookrightarrow \mathrm{Some}\, x * Q(x) \vee \ell \hookrightarrow - * \boxed{()}^{\gamma}}^{\iota}.$$

where we are using the exclusive resource algebra over the singleton set (see Example 7.18). The idea is that the spawn method returns this handle, which has information about the state of the forked-off thread. In the beginning the invariant is in the first state, with the location ℓ pointing to None. Once the spawn terminates the invariant will intuitively be in the second state, with $\ell \hookrightarrow \mathrm{Some}\, x$ and $Q(x)$.

When the join method is waiting for the location to become $\mathrm{Some}\, x$ it will have the $\boxed{()}^{\gamma}$ token, allowing it to know that the invariant is either in the first or the second state. Once it is in the second the join method will close the invariant in the third state, transferring $\boxed{()}^{\gamma}$ into it, and obtaining $Q(x)$.

Exercise 7.52. Following the description above show the following specifications for the spawn and join methods in detail.

$$\textsc{Ht-spawn}\quad \frac{\{P\}\, f\,()\, \{v.Q\}}{\{P\}\, \mathsf{spawn}(f)\, \{\ell.\exists \gamma.\, \mathrm{joinHandle}(\ell, \gamma, v.Q)\}} \qquad \textsc{Ht-join}\quad \frac{}{\{\mathrm{joinHandle}(\ell, \gamma, v.Q)\}\, \mathsf{join}(\ell)\, \{v.Q\}}$$

Using these specifications derive the Ht-par specification of the parallel composition operation.

◊

7.9 Case Study: Invariants for Sequential Programs

In this section we will use a technique known as Landin's knot to define recursive functions without the use of the primitive rec f(x) = e. Instead we are going to use *recursion through the store*. We first define myrec as:

$$\mathsf{myrec}\, f\ = \mathsf{let}\, r = \lambda x.\, x\, \mathsf{in}\, r \leftarrow (\lambda x.\, f\,(!\, r)\, x);\ !\, r$$

Before giving a specification to myrec we will try to get an intuition of how Landin's knot works. To this end we consider what a client implementing the factorial of a natural number could look like and how it would compute the factorial of 2. First we define the f we are going to use as:

$$\mathsf{F} = \lambda f.\lambda x.\mathsf{if}\, x = 0\, \mathsf{then}\, 1\, \mathsf{else}\, x * (f\,(x-1))$$

Next we define

$$\mathsf{fac} = \mathsf{myrec}\, \mathsf{F}$$

Hence we have that

$$\mathsf{fac}\, 2 = \mathsf{let}\, r = \lambda x.\, x\, \mathsf{in}\, r \leftarrow (\lambda x. \mathsf{F}\, (!\, r)\, x);\ (!\, r)\, 2$$

Now we consider how fac 2 reduces. After some steps, we have that $r \hookrightarrow (\lambda x. \mathsf{F}\,(!\,r)\,x)$ and the computation continues as

$$\begin{aligned}
(!\,r)\,2 &\rightsquigarrow (\lambda x. \mathsf{F}\,(!\,r)\,x)\ 2 \\
&\rightsquigarrow \mathsf{F}\,(!\,r)\,2 \\
&\rightsquigarrow \mathsf{F}\ (\lambda x. \mathsf{F}\,(!\,r)\,x)\ 2 \\
&\rightsquigarrow \lambda x. \mathsf{if}\, x = 0\, \mathsf{then}\, 1\, \mathsf{else}\, x * ((\lambda x. \mathsf{F}\,(!\,r)\,x)\ (x-1))\ 2 \\
&\rightsquigarrow \mathsf{if}\, 2 = 0\, \mathsf{then}\, 1\, \mathsf{else}\, 2 * ((\lambda x. \mathsf{F}\,(!\,r)\,x)\ (2-1)) \\
&\rightsquigarrow 2 * ((\lambda x. \mathsf{F}\,(!\,r)\,x)\ (2-1)) \\
&\rightsquigarrow 2 * ((\lambda x. \mathsf{F}\,(!\,r)\,x)\ 1) \\
&\vdots \\
&\rightsquigarrow 2 * \mathsf{if}\, 1 = 0\, \mathsf{then}\, 1\, \mathsf{else}\, 1 * ((\lambda x. \mathsf{fac}\,(!\,r)\,x)\ (1-1)) \\
&\vdots \\
&\rightsquigarrow 2 * 1 * ((\lambda x. \mathsf{F}\,(!\,r)\,x)\ 0) \\
&\rightsquigarrow 2 * ((\lambda x. \mathsf{F}\,(!\,r)\,x)\ 0) \\
&\rightsquigarrow 2 * (\mathsf{F}\,(!\,r)\,0) \\
&\vdots \\
&\rightsquigarrow 2 * \mathsf{if}\, 0 = 0\, \mathsf{then}\, 1\, \mathsf{else}\, 0 * (\lambda x. \mathsf{F}\,(!\,r)\,x)\ (0-1) \\
&\rightsquigarrow 2 * 1 \\
&\rightsquigarrow 2
\end{aligned}$$

Now it should be clear that the trick is that r is mapped to a λ-abstraction, *which refers to r in it's body*. Another important observation is that the value r points to never changes after it has been updated to $\lambda x. \mathsf{F}\,(!\,r)\,x$ in the beginning.

The specification we want myrec to satisfy is:

$$\textsc{Ht-myrec}\ \frac{\Gamma, f : Val, g : Val \mid S \land \forall y. \forall v. \{P\}\, g v\, \{u.Q\} \vdash \forall y. \forall v. \{P\}\, f\, g\, v\, \{u.Q\}}{\Gamma \mid S \vdash \forall y. \forall v. \{P\}\, \mathsf{myrec}\, f\, v\, \{u.Q\}}$$

This specification is similar to the first recursive rule from Section 4. The only differences is that here f is assumed to be a value and that we have $f\, g\, v$ in the premise instead of $e[g/f, v/x]$.

Proof of the specification for myrec

To prove Ht-myrec we assume the premise

$$\Gamma, f : Val, g : Val \mid S \land \forall y. \forall v. \{P\}\, g v\, \{u.Q\} \vdash \forall y. \forall v. \{P\}\, f\, g\, v\, \{u.Q\} \tag{31}$$

holds. By Ht-persistently and $\forall$I we have that

$$\Gamma, f : Val \mid S \vdash \forall g. \forall y. \forall v. \{P * (\forall y. \forall v. \{P\}\, g v\, \{u.Q\})\}\, f\, g\, v\, \{u.Q\} \tag{32}$$

holds.

The Hoare-triple we want to prove is

$$\forall y, \forall v. \{P\}\,(\lambda f.\,\mathsf{let}\, r = \lambda x.\, x \,\mathsf{in}\, r \leftarrow (\lambda x.\, f\,(!\,r)\,x);\ !\,r)\ f\ v\,\{u.Q\}$$

We proceed by using $\forall$I twice, Ht-beta, Ht-let, Ht-frame and Ht-alloc. Hence it suffices to prove

$$\forall r.\{\exists l.P * l = r * l \hookrightarrow (\lambda x.\, x)\}\, r \leftarrow (\lambda x.\, f\,(!\,r)\,x);\ (!\,r)\,v\,\{u.Q\}$$

By $\forall$I, Ht-exist, $\forall$I and Ht-Pre-Eq it suffices to show

$$\{P * l \hookrightarrow (\lambda x.\, x)\}\, l \leftarrow (\lambda x.\, f\,(!\,l)\,x);(!\,l)\,v\,\{u.Q\}$$

Now by Ht-seq, Ht-frame and Ht-store it suffices to show

$$\{P * l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)\}\,(!\,l)\,v\,\{u.Q\}$$

It the follows by Ht-bind-det and Ht-load that it suffices to show

$$\{P * l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)\}\,(\lambda x.\, f\,(!\,l)\,x)\ v\,\{u.Q\} \tag{33}$$

Thus by Ht-beta, Ht-bind-det and Ht-load it suffices to show

$$\{P * l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)\}\, f\ (\lambda x.\, f\,(!\,l)\,x)\,v\,\{u.Q\} \tag{34}$$

How do we proceed from here?

First attempt – "Naive approach" Looking at triple (34) and comparing it to the triple at the right hand side of the $\vdash$ in (31), we see that they are similar except:

1. the precondition of the current triple is $P * l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)$ instead of just P
2. in (34) we have $\lambda x.\, f\,(!\,l)\,x$ instead of g.

Regarding 1: As $P * l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x) \Rightarrow P$, it follows by Ht-csq that in order to prove a triple with precondition $P * l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)$ it suffices to prove the same triple with precondition P instead.

Regarding 2: As the g is actually quantified by a $\forall$[10], we have that if we can show that $\forall y. \forall v. \{P\}\,(\lambda x.\, f\,(!\,l)\,x)\,v\,\{u.Q\}$ holds then $\{P\}\, f\ (\lambda x.\, f\,(!\,l)\,x)\,v\,\{u.Q\}$ follows by the assumption that (31) holds.

Combining the two we get that it suffices to prove

$$\forall y. \forall v. \{P\}\,(\lambda x.\, f\,(!\,l)\,x)\,v\,\{u.Q\} \tag{35}$$

We proceed by using $\forall$I twice and Ht-beta. It thus suffices to prove

$$\{P\}\, f\,(!\,l)\,v\,\{u.Q\}$$

Now by Ht-bind we need to show

1. $f\ -\ v$ is an Evaluation context.
2. $\{P\}\,!\,l\,\{w.R\}$
3. $\forall w. \{R\}\, f\ w\ v\,\{u.Q\}$

for some R of our choice. Item (1) is clear since f is assumed to be a value. For item (2) we get stuck, as we no longer have $l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)$ in our precondition and thus we can not dereference l.

[10] see (32).

Second attempt – "Löb induction" In the beginning of Section 5, a fixed-point combinator for defining recursive functions was defined. In the proof of the rule for this fixed-point combinator the Löb rule played a crucial role. As we are doing something similar it might be useful to apply Löb induction here as well.

An immediate challenge is then: when should we apply the rule? The rule needs to be applied at a stage, which we will return to later, such that we may use the assumption we get at this later time. The two best places seem to be either where we ended (34)

$$\{P * l \hookrightarrow (\lambda x. f\,(!\,l)\,x)\}\, f\;(\lambda x. f\,(!\,l)\,x)\,v\,\{u.Q\}$$

or one step further back (33)

$$\{P * l \hookrightarrow (\lambda x. f\,(!\,l)\,x)\}\;(\lambda x. f\,(!\,l)\,x)\;v\,\{u.Q\}$$

Clearly, we won't return to the exact same code later (if this was the case, then we would loop indefinitely). When we return to either of these stages, the value v will be different, hence we of course want our Löb induction hypothesis to work for all values, not just a particular one.

In the second case this poses no problem at all; we added v to the context when we used ∀I in the beginning, hence we can use ∀E before using Löb induction to get a sufficiently general assumption.

In the first case, we can do the same trick for the value v (the second argument), but we can't do so for the first argument of f. Looking at item (3) in our first attempt, we see that it seems likely that we would need to consider a general value for the first argument to f as well.

Therefore, we decide to use Löb induction in the second case.

We start from (33). By ∀E it suffices to prove

$$\forall v. \{P * l \hookrightarrow (\lambda x. f\,(!\,l)\,x)\}\;(\lambda x. f\,(!\,l)\,x)\;v\,\{u.Q\}.$$

Hence by Löb induction we may assume that $\triangleright(\forall v. \{P * l \hookrightarrow (\lambda x. f\,(!\,l)\,x)\}\;(\lambda x. f\,(!\,l)\,x)\;v\,\{u.Q\})$ holds.

First we get rid of the ∀ quantification by using ∀I. We then proceed by Ht-beta, Ht-bind-det and Ht-load and thus it suffices to show

$$\{P * l \hookrightarrow (\lambda x. f\,(!\,l)\,x)\}\, f\;(\lambda x. f\,(!\,l)\,x)\,v\,\{u.Q\}$$

Now we proceed as in our first attempt. When we reach the point where we need to prove (35), we are almost able to conclude the proof. We need to prove

$$\forall y. \forall v. \{P\}\;(\lambda x. f\,(!\,l)\,x)\;v\,\{u.Q\}$$

and we have the assumption that[11]

$$\forall v. \{P * l \hookrightarrow (\lambda x. f\,(!\,l)\,x)\}\;(\lambda x. f\,(!\,l)\,x)\;v\,\{u.Q\}$$

The $\forall y$ we can get rid of by ∀I, but the assumption is not strong enough, as it requires us to have $l \hookrightarrow (\lambda x.\, f\,(!\,l)\,x)$ in the precondition, which we don't have. We are thus stuck again.

[11] Notice that the ▷ has been stripped from the assumption we got by the Löb induction. The reason for this is that the operational semantics has taken at least one step. Concretely, one could reason as follows; the assumption is persistent, as Hoare-triples are, and since ▷ commutes with □. We can therefore use Ht-persistently to move the assumption into the precondition of the Hoare-triple. Each time the operational semantics takes a step, we may use Ht-bind to bind the atomic step, and then use Ht-frame-atomic to remove the ▷ from the assumption. As the assumption without the later is still persistent, we may move it back into our assumptions by using Ht-persistently in the other direction.

If we instead had forgotten $l \hookrightarrow (\lambda x. f\,(!\,l)\,x)$, before using Löb induction, then we would have gotten a stronger assumption that doesn't require $l \hookrightarrow (\lambda x. f\,(!\,l)\,x)$ to be in the precondition. Our problem would then be, that giving up $l \hookrightarrow (\lambda x. f\,(!\,l)\,x)$, we almost immediately get stuck: we could go from

$$\{P\}\,(\lambda x. f\,(!\,l)\,x)\,v\,\{u.Q\}$$

to

$$\{P\}\,f\,(!\,l)\,v\,\{u.Q\}$$

but then the loss of the right to dereference l stops us from going any further.

Third attempt – "Invariants" When computing the factorial of 2, we noticed that the location kept pointing to the same value after it had been initialized and updated once, i.e., that it was invariant from this point on and onwards. Let us therefore try to state this as an invariant and see how things go. For this attempt, we will try to see if we can succeed without using Löb induction. Let $\mathcal{E}$ be the current mask[12].

As we have that $l \hookrightarrow (\lambda x. f\,(!\,l)\,x)$ and hence clearly $\triangleright(l \hookrightarrow (\lambda x. f\,(!\,l)\,x))$ it suffices by Ht-inv-alloc to prove

$$\exists\iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!\,l)\,x)}^{\iota} \vdash \{P\}\,f\,(\lambda x. f\,(!\,l)\,x)\,v\,\{u.Q\}_{\mathcal{E}}$$

We proceed as in the first attempt. When we get to the place, where we got stuck before, we can now use Ht-bind-det instead of Ht-bind, as this time we know, what the result of dereferencing l is. We thus need to show

1. $f\,-\,v$ is an evaluation context.
2. $\exists\iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!\,l)\,x)}^{\iota} \vdash \{P\}\,!\,l\,\{w.w = z \wedge R\}_{\mathcal{E}}$
3. $\exists\iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!\,l)\,x)}^{\iota} \vdash \{R[z/w]\}\,f\,z\,v\,\{u.Q\}_{\mathcal{E}}$

for some z and R of our choice. We consider each item in turn:

1. As before in the first attempt.
2. As $!\,l$ is atomic, we may use Ht-inv-open and thus it suffices to show

$$\exists\iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!\,l)\,x)}^{\iota} \vdash \begin{array}{l}\{P * \triangleright(l \hookrightarrow (\lambda x. f\,(!\,l)\,x))\}\\ \quad !\,l\\ \{w.w = (\lambda x. f\,(!\,l)\,x) \wedge R * \triangleright(l \hookrightarrow (\lambda x. f\,(!\,l)\,x))\}_{\mathcal{E}\setminus\{\iota\}}\end{array}$$

   Choosing R to be P, this triple then follows by Ht-frame and Ht-load.

3. Given the choice of R in item (2), the triple we need to prove is:

$$\exists\iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!\,l)\,x)}^{\iota} \vdash \{P\}\,f\,(\lambda x. f\,(!\,l)\,x)\,v\,\{u.Q\}_{\mathcal{E}}$$

   Now we are stuck again. f is a value and w is a value, hence the next reduction step would be to apply f to w, but we don't have any information about what then happens.

[12]We didn't explicitly annotate the Hoare-triples with masks before, as we didn't use any invariants.

Fourth attempt – "Invariants + Löb induction" Before beginning our fourth attempt, let us try to examine our previous attempts to see if they might give any clues about how to proceed.

In the first attempt we more or less just tried to brute-force the proof; this did not work out.

In the second attempt we tried to use Löb induction; this did not work either. The assumption, we got to deal with the recursive call was either too weak or came at the cost of the resource $l \hookrightarrow (\lambda x. f\,(!l)\,x)$, which was needed in order to get to the point, where the assumption had to be used.

In the third attempt, we proceeded mostly as in the first attempt, but tried to be a bit more clever by using an invariant to maintain the resource $l \hookrightarrow (\lambda x. f\,(!l)\,x)$. We got a bit further than in the first attempt, but we still did not have any way to deal with the recursive call.

From this it seems that we had two main challenges:

1. Dealing with the recursive call.
2. Maintaining the resource $l \hookrightarrow (\lambda x. f\,(!l)\,x)$.

From attempt 2 it seems that Löb induction might be able to deal with the recursive call, and from attempt 3 it seems that using an invariant might be sufficient to maintain the resource.

For our fourth attempt, we therefore decide to try using both of these techniques. Starting at the same point as in the second attempt, we have to prove

$$\{P * l \hookrightarrow (\lambda x. f\,(!l)\,x)\}\,(\lambda x. f\,(!l)\,x)\,v\,\{u.Q\}_{\mathcal{E}}$$

First we allocate the invariant by Ht-inv-alloc and hence we need to show

$$\exists \iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!l)\,x)}^{\iota} \vdash \{P\}\,(\lambda x. f\,(!l)\,x)\,v\,\{u.Q\}_{\mathcal{E}}$$

Now we use ∀E and then Löb induction. Thus we may assume

$$\triangleright(\forall v. \{P\}\,(\lambda x. f\,(!l)\,x)\,v\,\{u.Q\}_{\mathcal{E}})$$

By reasoning as in the third attempt, including opening the invariant in order to dereference l, we obtain that it suffices to show

$$\exists \iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!l)\,x)}^{\iota} \vdash \forall y. \forall v. \{P\}\,(\lambda x. f\,(!l)\,x)\,v\,\{u.Q\}_{\mathcal{E}}$$

This triple now follows by using ∀I to instantiate $\forall y$ and then using the assumption we got by the Löb induction. So, finally, we managed to prove the desired specification for myrec.

Exercise 7.53. Is it possible to prove the specification if one instead used Löb induction on the other case discussed in the second attempt? That is, can one prove

$$\exists \iota \in \mathcal{E}. \boxed{l \hookrightarrow (\lambda x. f\,(!l)\,x)}^{\iota} \vdash \forall v. \{P\}\,f\,(\lambda x. f\,(!l)\,x)\,v\,\{u.Q\}_{\mathcal{E}}$$

under the assumption of (31) and

$$\triangleright(\forall v. \{P\}\,f\,(\lambda x. f\,(!l)\,x)\,v\,\{u.Q\}_{\mathcal{E}})$$

If this is the case, then prove it. ◊

Client: fac

In this subsection we prove that the client fac does indeed compute the factorial of its argument. Recall the definition of fac

$$\mathsf{fac} = \mathsf{myrec}\,\mathsf{F}$$

where

$$\mathsf{F} = \lambda f.\lambda x.\mathsf{if}\, x = 0\,\mathsf{then}\, 1\,\mathsf{else}\, x * (f\,(x-1))$$

For the specification of fac we will use the standard mathematical definition of the factorial of a natural number:

$$\begin{aligned}\text{factorial } 0 &\equiv 1\\ \text{factorial}\,(n+1) &\equiv (n+1) * \text{factorial } n\end{aligned}$$

Now we can state the specification of fac as

$$\forall n.\,\{n \geq 0\}\,\mathsf{fac}\,n\,\{v.v = \text{factorial } n\}$$

Proof of the specification of fac Using Ht-myrec it suffices to show:

$$g : Val \mid \forall n.\,\{n \geq 0\}\,g\,n\,\{v.v = \text{factorial } n\} \vdash \forall n.\,\{n \geq 0\}\,\mathsf{F}\,g\,n\,\{v.v = \text{factorial } n\}$$

Hence we assume

$$\forall n.\,\{n \geq 0\}\,g\,n\,\{v.v = \text{factorial } n\}$$

and then we need to show

$$\forall n.\,\{n \geq 0\}\,\mathsf{F}\,g\,n\,\{v.v = \text{factorial } n\}$$

We proceed by using ∀I to instantiate n. Then we apply Ht-beta twice in order to apply F to its arguments. It thus suffices to prove

$$\{n \geq 0\}\,\mathsf{if}\,n = 0\,\mathsf{then}\,1\,\mathsf{else}\,n * (g\,(n-1))\,\{v.v = \text{factorial } n\}$$

Exercise 7.54. Complete the proof by proving the above triple. ◊

8 Case Study: Stacks with Helping

In this section we describe 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, moreover, 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 11.3 we discuss a specification technique, which can be used to give a stronger specification, which allows us to keep more precise track of the elements being pushed and popped.[13]

[13]Such a stronger specification can be found in `https://iris-project.org/pdfs/2017-case-study-concurrent-stacks-with-helping.pdf`.

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*.[14]

8.1 Mailboxes for Offers

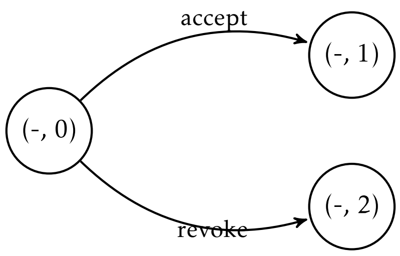
As described above we will use a side-channel in the stack implementation. This side-channel can be implemented and specified separately, and this is what we do now. A side-channel has the following operations:

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 three 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.

[14]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 code of the three methods is as follows.

$$\text{mk\_offer} \triangleq \lambda v.(v, \text{ref}(0))$$

$$\begin{aligned}\text{revoke\_offer} \triangleq \lambda v.\ & \text{let}\, u = \pi_1\, v\, \text{in}\\ & \text{let}\, s = \pi_2\, v\, \text{in}\\ & \text{if}\, \text{cas}(s, 0, 2)\, \text{then Some}\, u\, \text{else None}\end{aligned}$$

$$\begin{aligned}\text{accept\_offer} \triangleq\ & \text{let}\, u = \pi_1\, v\, \text{in}\\ & \text{let}\, s = \pi_2\, v\, \text{in}\\ & \text{if}\, \text{cas}(s, 0, 1)\, \text{then Some}\, u\, \text{else None}\end{aligned}$$

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.

$$\text{mailbox} \triangleq \lambda\_.\, \text{let}\, r = \text{ref(None)}\, \text{in}$$
$$\left(\left(\begin{aligned}\lambda v.\quad & \text{let off} = \text{mk\_offer}\, v\, \text{in}\\ & r \leftarrow \text{Some off}; \text{revoke\_offer off}\end{aligned} \right), \left(\begin{aligned}\lambda\_.\quad & \text{let offopt} = !\, r\, \text{in}\\ & \text{match offopt with}\\ & \quad \text{None} \quad \Rightarrow \text{None}\\ & \mid \text{Some}\, x \Rightarrow \text{accept\_offer}\, x\\ & \text{end}\end{aligned} \right) \right)$$

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.

8.2 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)
```

8.3 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 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.

8.4 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.

$$\forall \gamma, v.\, \{\mathsf{is\_offer}_{\gamma}(v)\}\, \mathsf{accept\_offer}(v) \{u. u = \mathrm{None} \vee \exists v'.\, u = \mathrm{Some}(v') * \Phi(v')\}$$

Proofs of these specifications are entirely straightforward based on what we have seen up until now, so we leave them to the reader.

8.5 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.

$$\{\mathsf{True}\}\ \mathsf{mailbox}()\left\{u.\ \begin{array}{l}\exists\,\mathsf{put}\,\mathsf{get}.\\ u = (\mathsf{put},\mathsf{get}) * \\ \forall v.\,\{\Phi(v)\}\ \mathsf{put}(v)\,\{w. w = \mathsf{None} \vee \exists v'.\, w = \mathsf{Some}(v') * \Phi(v')\} * \\ \{\mathsf{True}\}\ \mathsf{get}()\,\{w. w = \mathsf{None} \vee \exists v'.\, w = \mathsf{Some}(v') * \Phi(v')\}\end{array}\right\} \tag{36}$$

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

$$\mathsf{is\_mailbox}(\ell) \triangleq \ell \hookrightarrow \mathsf{None} \vee \exists v \gamma.\, \ell \hookrightarrow \mathsf{Some}(v) * \mathsf{is\_offer}_{\gamma}(v)$$

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.

8.6 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.

$$\forall \Phi.\{\mathsf{True}\}\ \mathsf{stack}()\left\{p.\exists\,\mathsf{pop}\,\mathsf{push}.\ \begin{array}{l} p = (\mathsf{pop},\mathsf{push}) * \\ \{\mathsf{True}\}\ \mathsf{pop}()\,\{v. v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}\, v' * \Phi(v')\} * \\ \forall v.\,\{\Phi(v)\}\ \mathsf{push}\, v\,\{u. u = () * \mathsf{True}\}\end{array}\right\} \tag{37}$$

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

$$\mathsf{is\_stack}(v) \triangleq v = \mathsf{None} \vee \exists h, t.\, v = \mathsf{Some}(h,t) * \Phi(h) * \triangleright \mathsf{is\_stack}(t)$$

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.

$$\mathsf{stack\_inv}(\ell) \triangleq \exists v'.\, \ell \hookrightarrow v' * \mathsf{is\_stack}(v')$$

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

$$\{r \hookrightarrow \mathsf{None}\}\,(\mathsf{pop},\mathsf{push})\left\{p.\exists\,\mathsf{pop}\,\mathsf{push}.\ \begin{array}{l} p = (\mathsf{pop},\mathsf{push}) * \\ \{\mathsf{True}\}\ \mathsf{pop}()\,\{v. v = \mathsf{None} \vee \exists v'.\, v = \mathsf{Some}\, v' * \Phi(v')\} * \\ \forall v.\,\{\Phi(v)\}\ \mathsf{push}\, v\,\{u. u = () * \mathsf{True}\}\end{array}\right\}$$

where $(\mathsf{pop},\mathsf{push})$ are the two methods in the body of the stack method. We should show this in a context where we have

$$\begin{array}{l}\forall v.\,\{\Phi(v)\}\ \mathsf{put}(v)\,\{w. w = \mathsf{None} \vee \exists v'.\, w = \mathsf{Some}(v') * \Phi(v')\}\\ \{\mathsf{True}\}\ \mathsf{get}()\,\{w. w = \mathsf{None} \vee \exists v'.\, w = \mathsf{Some}(v') * \Phi(v')\},\end{array}$$

the specification of the mailbox. Before verifying the specifications we allocate an invariant containing $\mathsf{stack\_inv}$ which we can, since $r \hookrightarrow \mathsf{None}$ implies $\mathsf{stack\_inv}$. 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 with $\mathsf{stack\_inv}$ 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

$$\triangleright\{\mathsf{True}\}\ \mathrm{pop}()\ \{v.v = \mathrm{None} \vee \exists v'. v = \mathrm{Some}\, v' * \Phi(v')\}.$$

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 $\mathsf{stack\_inv}$ to read $!r$, after which, by using the definition of the $\mathsf{stack\_inv}$ predicate, we have to consider two cases. If the stack is empty ($r \hookrightarrow \mathsf{None}$) 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 $\triangleright\mathsf{stack\_inv}(t)$. To proceed we need to open the invariant again, and after that we again consider two cases.

- If now $r \hookrightarrow \mathsf{None}$, then cas fails and we simply use the Löb induction hypothesis to proceed.
- Otherwise, $r \hookrightarrow \mathsf{Some}(h', t')$, where $\Phi(h')$ and $\triangleright\mathsf{stack\_inv}(t')$ 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 $\mathsf{stack\_inv}(t)$ holds. Otherwise cas fails and we are again done by the Löb induction hypothesis.

9 Case Study: Ticket Lock

We have seen the specification of a basic spin lock in Section 7.6. However, the basic spin lock easily leads to thread starvation. A thread can be indefinitely denied access to the critical region protected by the lock even under very strong fairness assumptions on the scheduler. This might be acceptable in certain applications, but is undesirable in others. A *ticket lock* is a more refined spin lock which keeps track of the order in which access to critical region was demanded. It does this by keeping two counters which represent queue positions. The first counter is the queue position that is next in line for access to the critical region, and the second is the position of the client in the queue. When acquiring the lock, the client first obtains a position in the queue (a *ticket*) by incrementing (atomically, using cas) the second counter. Then it waits until it is its turn for entering the critical region. When releasing the lock, the first counter is incremented to allow other waiting clients to acquire the lock.

This type of lock thus behaves like a *first-in first-out* queue. This means that the threads trying to acquire the lock will be allowed to take it in the order they asked for it, or more precisely, in the order in which they obtained their tickets. However, the specification of the ticket lock we give in this section does not express this property. Indeed, we give the ticket lock the same specification as we gave to the spin lock in Example 7.36 on page 7.36. Fairness is a *liveness* property, and in fact a global property of the system, *i.e.*, not a property of an individual method, whereas Iris – as presented here – only allows reasoning about safety properties.

9.1 Setup

The ticket lock has three methods, newLock for creating a new ticket lock, and acquire and release for acquiring and releasing the lock respectively. Moreover, we have an auxiliary wait method used by the acquire method. In order to make the verification more readable we write it as a separate public method.

```
let newLock() = (ref(0), ref(0))
let acquire l = let n = !(π₂ l) in
                if cas((π₂ l), n, n + 1)
                then wait o l
                else acquire l
let wait n l = let o = !(π₁ l) in
               if o = n then () else wait n l
let release l = (π₁ l) ← !(π₁ l) + 1
```

As we explained above, the lock uses two mutable counters, the first for keeping track of who is currently permitted to take the lock and thereby gain access to the resources protected by it, and the second for tracking which ticket to give out to the next thread. We will refer to them as ***owner*** and ***next*** respectively.

In the acquire method we first try to obtain a ticket. This is the intention of the first two lines. However since another thread might try to obtain a ticket at the same time we need to use cas to atomically increment, and thus might need to retry this operation. Once we succeed, we obtain a ticket in the queue, and proceed to wait our turn to get access to the resources protected by the lock by using the wait method.

Remark 9.1. Note that it is indeed possible that we never obtain a ticket, since the scheduler might decide to schedule another thread every time just after we have read the value of the second counter. Thus we might always fail to acquire the lock. However, getting a ticket is a cheap operation, and we have to be exceedingly unlucky to always be preempted just after reading the counter. In contrast, threads can spend a significant amount of time in the critical region, hence the spin lock is more likely to fail and always be prevented from acquiring the lock.

In a real implementation we would use a primitive *fetch-and-add* operation supported by many modern processors instead of the cas loop. This would provide an even stronger guarantee. We have chosen to keep things simple in these notes and not complicate the language too much, and hence we stick with only the cas primitive. ■

The wait method *waits* until it is the given client's turn to enter the critical region. Note that the method does not need to use any synchronisation primitives since only the thread that acquired the lock should release it, and thus the value of n will not change once it becomes n; this is the invariant maintained by all the methods of the ticket lock module.

Analogously, we do not need to use synchronisation primitives when releasing the lock (the release method). Only the method which is in the critical region will increment the first counter, and thus there will be no interference between reading its value, and writing it.

As mentioned in the introduction, we can give this lock the same specification as the spin lock. The only addition is the specification of the wait method, together with the auxiliary assertion issued used by it.

$$\begin{aligned}
&\exists\, \mathsf{isLock} : \mathsf{GhostName} \to \mathit{Val} \to \mathsf{Prop} \to \mathsf{Prop}.\\
&\exists\, \mathsf{locked} : \mathsf{GhostName} \to \mathsf{Prop}.\\
&\exists\, \mathsf{issued} : \mathsf{GhostName} \to \mathit{Val} \to \mathsf{Prop}.\\
&\qquad \forall P, v, \gamma.\ \mathrm{isLock}(v, P, \gamma) \Rightarrow \Box \mathrm{isLock}(v, P, \gamma)\\
&\wedge \quad \forall \gamma.\ \mathrm{locked}(\gamma) * \mathrm{locked}(\gamma) \Rightarrow \mathsf{False}\\
&\wedge \quad \forall \gamma.\ \mathrm{issued}(\gamma, n) * \mathrm{issued}(\gamma, n) \Rightarrow \mathsf{False}\\
&\wedge \quad \forall P.\ \{P\}\ \mathrm{newLock}()\ \{v. \exists \gamma.\ \mathrm{isLock}(v, P, \gamma)\}\\
&\wedge \quad \forall P, v, \gamma, n.\ \{\mathrm{isLock}(v, P, \gamma) * \mathrm{issued}(\gamma, n)\}\ \mathrm{wait}(n, v)\ \{\_.P * \mathrm{locked}(\gamma)\}\\
&\wedge \quad \forall P, v, \gamma.\ \{\mathrm{isLock}(v, P, \gamma)\}\ \mathrm{acquire}(v)\ \{\_.P * \mathrm{locked}(\gamma)\}\\
&\wedge \quad \forall P, v, \gamma.\ \{\mathrm{isLock}(v, P, \gamma) * P * \mathrm{locked}(\gamma)\}\ \mathrm{release}(v)\ \{\_.\mathsf{True}\}
\end{aligned}$$

The abstract predicate isLock(v, P, γ) expresses that the value v (a pair of two locations) is a ticket lock protecting resources described by the predicate P using the ghost resources associated with γ. As before, the isLock predicate is persistent, so different threads can use the lock simultaneously.

We will start by defining the resource algebra, invariant, and predicates we need, and then explain our choices. The resource algebra we use is: $\textsc{Auth}(\mathrm{Ex}(\mathbb{N})_? \times (\mathbb{P}_{fin}(\mathbb{N}))$. All of the constructs of this RA have been previously introduced in Section 7.4. The lock invariant we will use in our proof is:

$$\begin{aligned}
\mathrm{lockInv}(\gamma, \ell_o, \ell_n, P) = \exists o, n.\ & \ell_o \hookrightarrow o * \ell_n \hookrightarrow n * \left[\bullet\, (o, \{i \mid 0 \le i < n\})\right]^{\gamma}\\
& * \left(\left[\circ\, (o, \emptyset)\right]^{\gamma} * P \vee \left[\circ\, (\varepsilon, \{o\})\right]^{\gamma}\right).
\end{aligned}$$

With this we define the isLock, locked and issued predicates as follows (where $\mathcal{E}$ is some chosen infinite set of invariant names)

$$\begin{aligned}
\mathrm{isLock}(v, P, \gamma) &= \exists \ell_o, \ell_n, \iota \in \mathcal{E}.\ v = (\ell_o, \ell_n) * \boxed{\mathrm{lockInv}(\gamma, \ell_o, \ell_n, P)}^{\iota}\\
\mathrm{locked}(\gamma) &= \exists o. \left[\circ\, (o, \emptyset)\right]^{\gamma}\\
\mathrm{issued}(\gamma, n) &= \left[\circ\, (\varepsilon, \{n\})\right]^{\gamma}
\end{aligned}$$

Our lock invariant keeps track of the values of the owner and next counters. Since these change when the lock is being used, they need to be existentially quantified. The exclusive construction in our RA expresses the fact that only one thread is allowed to take or hold the lock at any given time. Moreover, the invariant holds an authoritative piece of ghost state, which holds both the value of the owner counter and a subset of natural numbers. This set keeps track of which tickets have been given out. The combination of the authoritative construction and the exclusive one is needed in order to conclude equivalent values when combining different parts of our ghost state. The disjunction in our invariant intuitively describes whether the lock is currently acquired or not. If the left disjunct holds then the thread with ticket number o is allowed to take the lock, but has not yet done so. This is also the reason why the lock in this case holds the predicate P. If the right disjunct holds then the lock is held by the thread with ticket number o.

The locked (γ) predicate expresses that the lock associated with ghost name γ is currently locked with some ticket number. The predicate issued (γ, n) states that ticket number n has been given out.

These two predicates correspond to the two sides of the disjunction in the invariant. The intuition behind this is that when we know either locked or issued, then we can conclude that the other side of the disjunction must currently be in the invariant, since the composition would otherwise be invalid.

9.2 Proofs

There are five proof obligations in the specification. The first three are derived directly from the properties of the chosen resource algebra, and the definitions of the relevant predicates. The isLock predicate is persistent since it is a separating conjunction of two persistent propositions. The locked predicate is not duplicable due to the use of the exclusive part of our RA. Finally, the issued predicate is not duplicable, because the composition of two sets in our RA is only valid if these sets are disjoint.

9.2.1 newLock

The fourth obligation is the specification for newLock and will therefore include the allocation of ghost state and the lock invariant. We need to show the following triple:

$$\{P\}\ \text{newLock}()\ \{v.\exists\gamma.\ \text{isLock}(v,P,\gamma)\}.$$

By Ht-beta, is suffices to show:

$$\{P\}\ (\mathsf{ref}(0),\mathsf{ref}(0))\ \{v.\exists\gamma.\ \text{isLock}(v,P,\gamma)\}.$$

We first apply the Ht-bind-det rule twice with $\mathsf{ref}(0)$ in both cases. We then allocate the ghost state $\boxed{(\bullet(0,\emptyset))\cdot\circ(0,\emptyset)}^{\gamma}$. Keeping in mind that $\boxed{(\bullet(0,\emptyset))\cdot\circ(0,\emptyset)}^{\gamma} \vdash \boxed{\bullet(0,\emptyset)}^{\gamma} * \boxed{\circ(0,\emptyset)}^{\gamma}$, we get the two pieces of ghost state we need. Together with P from the precondition and the two locations we got from the two $\mathsf{ref}(0)$, we now have all the resources needed for the isLock predicate and are done.

9.2.2 wait

The next proof obligation is the specification for wait. We want to show:

$$\{\text{isLock}(v,P,\gamma) * \text{issued}(\gamma,n)\}\ \text{wait}(n,v)\ \{\_.P * \text{locked}(\gamma)\}$$

This is a recursive definition, so we make use of the derived rule from Exercise 6.4 together with the Ht-beta and Ht-csq rules. This means we assume:

$$\{\triangleright\,\text{isLock}(v,P,\gamma) * \triangleright\,\text{issued}(\gamma,n)\}\ \text{wait}(n,v)\ \{\_.P * \text{locked}(\gamma)\}$$

and need to show

$$\{\text{isLock}(v,P,\gamma) * \text{issued}(\gamma,n)\}\ \mathsf{let}\, o = !(\pi_1\, v)\, \mathsf{in} \ldots\ \{\_.P * \text{locked}(\gamma)\}$$

By unfolding the isLock predicate, we get the concrete pair of locations ℓ_o, ℓ_n we can substitute v with. Moreover, we can move our invariant into the context. We will refer to $\text{lockInv}(\gamma,\ell_o,\ell_n,P)$ as I for the remainder of the proof in order to keep the notation shorter. We now use the Ht-Proj rule followed by the Ht-let-det and the Ht-bind-det rules with the intermediate postcondition $\{w.(w = n * \text{locked}(\gamma) * P) \vee (w \neq n * \text{issued}(\gamma,n))\}$. So we first need to see what $!\ell_o$ evaluates to. In order to do this, we need to open our invariant with the Ht-inv-open rule, since it holds the

$\ell_o \hookrightarrow o$ information we need. We then proceed by casing on whether $n = o$. In the case where $n \neq o$, we need to show:

$$\boxed{I}^{\iota} \vdash \{\triangleright I * n \neq o * \text{issued}(\gamma, n)\}\, !\,\ell_o\, \{w.((w = n * \text{locked}(\gamma) * P) \lor (w \neq n * \text{issued}(\gamma, n))) * \triangleright I\}$$

We can now use the Ht-load rule and choose to prove the right side of the disjunction in the postcondition, since we directly get this information from our precondition. In the case where $n = o$, we need to show:

$$\boxed{I}^{\iota} \vdash \{\triangleright I * n = o * \text{issued}(\gamma, n)\}\, !\,\ell_o\, \{w.((w = n * \text{locked}(\gamma) * P) \lor (w \neq n * \text{issued}(\gamma, n))) * \triangleright I\}$$

Again we start with the Ht-load rule. In this case, we need to prove the left side of the disjunction. We can now unpack issued and I and replace n with o. The composition of the different pieces of ghost state we now own is only valid if the left side of the disjunction in the precondition is currently true, i.e. we get:

$$\ell_o \hookrightarrow o * \ell_n \hookrightarrow n * \boxed{\bullet(o, \{i \mid 0 \leq i < n\})}^{\gamma} * \boxed{\circ(o, \emptyset)}^{\gamma} * P * \boxed{\circ(\varepsilon, \{o\})}^{\gamma}$$

We can now reestablish the invariant, this time with the resources for the right side of the disjunction. This leaves us with: $\boxed{\circ(o, \emptyset)}^{\gamma} * P$, which is exactly what we need for the postcondition $\text{locked}(\gamma) * P$.

The remaining proof obligation at this point is:

$$\boxed{I}^{\iota} \vdash \{((o = n * \text{locked}(\gamma) * P) \lor (o \neq n * \text{issued}(\gamma, n)))\}\, \textbf{if}\, n = o\, \textbf{then}\, ()\, \textbf{else}\, \text{wait}\, n\, l\, \{\_.P * \text{locked}(\gamma)\}$$

Since our precondition is a disjunction, we will need to show the triple twice assuming the left and the right side respectively. If we assume the left side of the disjunction we need to show:

$$\boxed{I}^{\iota} \vdash \{o = n * \text{locked}(\gamma) * P\}\, \textbf{if}\, n = o\, \textbf{then}\, ()\, \textbf{else}\, \text{wait}\, n\, l\, \{\_.P * \text{locked}(\gamma)\}$$

Since our precondition gives us the result of the if statement, we can directly use Ht-If-True and notice that the postcondition follows directly from our assumptions, so we are done. Assuming the right side, we have to show:

$$\boxed{I}^{\iota} \vdash \{o \neq n * \text{issued}(\gamma, n)\}\, \textbf{if}\, n = o\, \textbf{then}\, ()\, \textbf{else}\, \text{wait}\, n\, l\, \{\_.P * \text{locked}(\gamma)\}$$

Again, our precondition gives us the result to the if, so we can use the Ht-If-False rule and end up with having to reason about the recursive call to wait. The triple matches our outermost assumption, so we apply the induction hypothesis (*i.e.*, the assumption of the premise of the rule in Exercise 6.4) and are done.

9.2.3 acquire

With the specification for wait in place, we are now able to show the one for acquire. We need to show the following triple:

$$\{\text{isLock}(v, P, \gamma)\}\ \text{acquire}(v)\ \{\_.P * \text{locked}(\gamma)\}$$

Since acquire uses wait internally, the proof of this specification will make use of the one for wait. Notice how the postconditions of both specifications are exactly the same. This means our proof intuitively consists of getting from our current precondition to the one for wait at the point where we call wait, since we can then apply its specification and be done.

We begin the same way as with wait, since we are dealing with a recursive definition again. So we assume:

$$\{\triangleright\,\mathrm{isLock}(v,P,\gamma)\}\ \mathrm{acquire}(v)\,\{\_.P * \mathrm{locked}(\gamma)\}$$

and want to show:

$$\{\mathrm{isLock}(v,P,\gamma)\}\,\mathsf{let}\,o = !(\pi_2\,v)\,\mathsf{in}\ldots\{\_.P * \mathrm{locked}(\gamma)\}$$

By unfolding the isLock predicate, we get the concrete pair of locations ℓ_o, ℓ_n we can substitute v with. Moreover, we can move our invariant into the context. Now we use Ht-let-det twice and Ht-Proj once. At this point we have the expression $!\ell_n$, so we need to open the invariant in order to access the $\ell_n \hookrightarrow n$ information stored in it. Since we don't change anything, we can close the invariant again after getting the stored value n. This means we now need to show:

$$\boxed{\mathrm{lockInv}(\gamma,\ell_o,\ell_n,P)}^{\iota} \vdash \{\mathsf{True}\}\,\mathsf{if}\,\mathsf{cas}(\pi_2\,(\ell_o,\ell_n),o,o+1)\,\mathsf{then}\,\mathrm{wait}\,o\ v\,\mathsf{else}\,\mathrm{acquire}\,v\,\{\_.P * \mathrm{locked}(\gamma)\}$$

We use the Ht-Proj and Ht-op rules, followed by the bind rule with the intermediate goal.

$$\boxed{\mathrm{lockInv}(\gamma,\ell_o,\ell_n,P)}^{\iota} \vdash \{\mathsf{True}\}\,\mathsf{cas}(\ell_n,o,o+1)\,\{u.(u = \mathsf{true} * \mathrm{issued}(\gamma,o)) \vee u = \mathsf{false})\}$$

Since cas is atomic, we can open our invariant again. We then proceed by casing on $n = o$. If $n = o$, then we can use the Ht-CAS-succ rule. By Ht-csq, it is sufficient to show either side of the disjunction in the postcondition. In this case, we choose $(u = \mathsf{true} * \mathrm{issued}(\gamma,o))$. We can now update our authoritative ghost state from $\bullet(o,\{i \mid 0 \le i < n\})$ to $\bullet(o,\{i \mid 0 \le i < n\} \cup \{n\}) \cdot \circ(\emptyset,\{n\})$. Performing the union gives us exactly the resources we need to close the invariant again. Moreover, the remaining ghost state $\circ(\emptyset,\{n\})$ is exactly $\mathrm{issued}(\gamma,n)$, so we are done with the intermediate goal. At this point we need to show:

$$\boxed{\mathrm{lockInv}(\gamma,\ell_o,\ell_n,P)}^{\iota} \vdash \{\mathrm{issued}(\gamma,n)\}\,\mathsf{if}\,\mathsf{true}\,\mathsf{then}\,\mathrm{wait}\,n\ v\,\mathsf{else}\,\mathrm{acquire}\,v\,\{\_.P * \mathrm{locked}(\gamma)\}$$

We therefore proceed with the Ht-If-True rule and end up exactly where we wanted to be in order to apply our wait specification, which takes care of the rest of this case.

In the case of $n \neq o$, we apply the Ht-CAS-fail rule and choose to prove the other side of the disjunction. We haven't changed anything and can therefore close our invariant again and are done with the intermediate goal. At this point, we need to show:

$$\boxed{\mathrm{lockInv}(\gamma,\ell_o,\ell_n,P)}^{\iota} \vdash \{\mathsf{True}\}\,\mathsf{if}\,\mathsf{false}\,\mathsf{then}\,\mathrm{wait}\,n\ v\,\mathsf{else}\,\mathrm{acquire}\,v\,\{\_.P * \mathrm{locked}(\gamma)\}$$

We therefore proceed with the Ht-If-False rule and notice that the remaining proof obligation matches the assumption about the recursive call, so we are done.

9.2.4 release

The final obligation is the specification for the release method. We need to show the following triple:

$$\{\mathrm{isLock}(v,P,\gamma) * \mathrm{locked}(\gamma) * P\}\ \mathrm{release}(v)\,\{\_.\mathsf{True}\}.$$

We begin with the Ht-beta rule as usual. Unfolding the isLock predicate in the precondition, we get two specific locations, ℓ_o and ℓ_n, and can therefore substitute v for the pair of these two. Then we make use of the Ht-Proj rule twice. At this point, we have to show:

$$\{\mathrm{lockInv}(\gamma,\ell_o,\ell_n,P) * \mathrm{locked}(\gamma) * P\}\,\ell_o \leftarrow !\ell_o + 1\,\{\_.\mathsf{True}\}.$$

We now apply the Ht-bind-det rule with $!\ell_o$. In order to reason about this expression, we need the information stored in our invariant. Since reading from a location is an atomic operation we can open our invariant at this point which will give us

$$\ell_o \hookrightarrow o * \ell_n \hookrightarrow n * \boxed{\bullet(o, \{i \mid 0 < i \leq n\})}^{\gamma} * ((\boxed{\circ(o, \emptyset)}^{\gamma} * P) \vee \boxed{\circ(\varepsilon, \{o\})}^{\gamma})$$

for some numbers o and n. So we can use the Ht-load rule now that we have $\ell_o \hookrightarrow o$ specifically.

Unfolding the locked(γ) predicate gives us $\boxed{\circ(o', \emptyset)}^{\gamma}$ for some number o'. Since the composition of this and the authoritative piece of ghost state must be valid, we get that $o = o'$ and we can therefore substitute one of them away. We now close our invariant again.

We then bind $o + 1$, use the Ht-op rule and are left with showing:

$$\left\{\text{lockInv}(\gamma, \ell_o, \ell_n, P) * \boxed{\circ(o, \emptyset)}^{\gamma} * P\right\} \ell_o \leftarrow (o + 1) \{\_.\mathsf{True}\}.$$

At this point, we can open the invariant again, since storing a value is atomic. As before, this gives us access to the needed location and we can therefore use the Ht-store rule. We moreover again conclude that the two o values in our ghost state must be the same. Apart from this, we can see that we have $\boxed{\circ(o, \emptyset)}^{\gamma}$, which means that the disjunction part of the invariant must hold $\boxed{\circ(\varepsilon, \{o\})}^{\gamma}$, since the composition would otherwise not be valid.

At this point we can update

$$\boxed{\bullet(o, \{i \mid 0 \leq i < n\})}^{\gamma} \cdot \boxed{\circ(o, \emptyset)}^{\gamma}$$

to

$$\boxed{\bullet((o + 1), \{i \mid 0 \leq i < n\})}^{\gamma} \cdot \boxed{\circ((o + 1), \emptyset)}^{\gamma},$$

and then we can close the invariant again with these two pieces of ghost state along with P.

9.3 Discussion

As mentioned earlier, the lock presented in this section is based on two counters. Earlier in the notes, we have already given different specifications for counters, but rather than utilizing these, the ticket lock implementation and proof refers to the implementation of the counters. Hence it is not modular — ideally, we should be able to verify the ticket lock relative to an abstract specification of the counter module. This point will be addressed further in the following Section 11.

10 Case study: The Array-Based Queueing Lock

The array-based queuing lock (ABQL) is a lock closely related to the ticket lock. The cache behaviour of the ABQL scales better to a large number of threads at the cost of increased space usage and of only being safe with a bounded number of concurrent threads. The latter must be modelled when proving safety of the lock, and hence the ABQL poses unique and interesting challenges to such efforts.

In this section we describe the array-based queuing lock and its implementation in $\lambda_{\text{ref,conc}}$. As we have done for the other locks, we then give and prove a specification which shows that the ABQL is safe, and hence that it satisfies mutual exclusion.

10.1 Arrays in $\lambda_{\text{ref,conc}}$

As the name of the array-based queuing lock implies, its implementation relies on arrays. We therefore first introduce how arrays are supported in $\lambda_{\text{ref,conc}}$.

One may wonder why arrays are necessary, given that we have already seen how lists can be implemented in $\lambda_{\text{ref,conc}}$. The key difference is that arrays offer constant time random-access to elements in the array. In addition to the performance benefits of this, it also means that reading and writing elements in an array is an atomic expression.

We need to be able to create arrays of a given size, access elements in an array at a specific index, and update values at an index in an array. To this end $\lambda_{\text{ref,conc}}$ includes the following syntactic constructs:

$$\begin{array}{llll} & \circledcirc & ::= & \cdots \mid +_l \\ \mathit{Exp} & e & ::= & \cdots \mid \mathsf{alloc}_N(e, e) \\ \mathit{ECtx} & E & ::= & \cdots \mid \mathsf{alloc}_N(E, e) \mid \mathsf{alloc}_N(v, E) \end{array}$$

In order to express that a location ℓ points to an array corresponding to a list v, we use the predicate $\ell \hookrightarrow_* v$ with the typing rule

$$\frac{\Gamma \vdash \ell : \mathsf{Val} \qquad \Gamma \vdash v : \mathsf{list\ Val}}{\Gamma \vdash \ell \hookrightarrow_* v : \mathsf{Prop}}$$

If n is a number and v is a value, then $\mathsf{alloc}_N(n, v)$ represents the allocation of an array that is n long and which is initialized with the value v at every index. We use $\mathrm{replicate}(n, v)$ to denote a mathematical list consisting of the value v repeated n times. The behavior of alloc_N is then captured by the following rule:

$$\textsc{Ht-allocN}\quad \frac{}{S \vdash \{\mathsf{True}\}\, \mathsf{alloc}_N(n, v)\, \{v.\exists \ell. v = \ell \wedge \ell \hookrightarrow_* \mathrm{replicate}(n, v)\}}$$

To index into an array the $+_l$ operator is used. If ℓ points to the start of an array then $\ell +_l i$ is a location pointing to the value of the i'th element, counting from 0. If xs is a mathematical list, we use the notation xs_i to denote the i'th entry in the list.

$$\textsc{Ht-load-offset}\quad \frac{}{S \vdash \{\ell \hookrightarrow_* xs * xs_i = e\}\, !(\ell +_l i)\, \{v.v = e \wedge \ell \hookrightarrow_* xs\}}$$

To update an element in an array we use the store operator in combination with $+_l$.

$$\textsc{Ht-store-offset}\quad \frac{}{S \vdash \{\ell \hookrightarrow_* xs\}\, (\ell +_l n) \leftarrow v\, \{v.v = () \wedge \ell \hookrightarrow_* xs[n := v]\}}$$

The operator $+_l$ can be thought of as pointer arithmetic and then this way of using arrays is similar to how arrays are implemented in many low-level programming languages. Handling arrays in this manner in Iris has the benefit that we only need to add $+_l$ and can reuse the load and store operators in the context of arrays.

10.2 The implementation

Having seen how arrays work, we now describe the ABQL and present an implementation in $\lambda_{\text{ref,conc}}$.

The lock consists of a natural number *next* and an array of booleans *array*. The number represents the next ticket available and the array contains false at every entry, except for one, which contains true. The *index* of the true value in the array represents which ticket currently grants access to the lock.

Constructing an ABQL takes as argument a natural number *cap*, specifying the capacity of the lock. An array of length *cap* is then initialized with true at the first entry and false at all other entries. The natural number which represents the next ticket is initially set to 0.

To acquire the lock, a ticket is received using faa. The primitive faa atomically returns the number stored at a location and adds a number to the stored value. The received ticket, modulo the capacity of the lock, is an index into the array. By spinning on the value at this index in the array until it becomes true, the lock is acquired.

To release the lock, the entry corresponding to the ticket from which the lock was acquired is updated from true to false. Then, the next index in the array, modulo *cap*, is set to true. This signals to the next thread that it can acquire the lock.

In the implementation we use a triple to represent the lock. We use $(-,-,-)$ as notation for $((-,-),-)$ and abuse notation slightly by letting π_1, π_2, and π_3 denote the projections of a triple denoted in this manner. The first element in the triple is the array, the second the *next* number, and the third is the length of the array, which we store since arrays in $\lambda_{\text{ref,conc}}$ do not have a length operation. The implementation is then:

$$
\begin{aligned}
\mathsf{let}\,\text{newLock}\,cap &= \mathsf{let}\,array = \mathsf{alloc}_N(\mathsf{false},\ cap)\,\mathsf{in}\\
&\quad (array +_l 0) \leftarrow \mathsf{true};\\
&\quad (array,\ \mathsf{ref}(0),\ cap)\\
\mathsf{let}\,\text{acquire}\ l &= \mathsf{let}\,next = \pi_2\ l\,\mathsf{in}\\
&\quad \mathsf{let}\,ticket = \mathsf{faa}\ next\ 1\,\mathsf{in}\\
&\quad \text{wait}\,l\ ticket;\\
&\quad ticket\\
\mathsf{let}\,\text{wait}\,l\ t &= \mathsf{let}\,array = \pi_1\ l\,\mathsf{in}\\
&\quad \mathsf{let}\,i = t\ \text{rem}\ (\pi_3\ l)\,\mathsf{in}\\
&\quad \mathsf{if}\ !(array +_l i)\,\mathsf{then}\,()\,\mathsf{else}\,(\text{wait}\,l\ t)\\
\mathsf{let}\,\text{release}\ l\ o &= \mathsf{let}\,array = \pi_1\ l\,\mathsf{in}\\
&\quad \mathsf{let}\,cap = \pi_3\ l\,\mathsf{in}\\
&\quad array +_l (o\ \text{rem}\ cap) \leftarrow \mathsf{false};\\
&\quad array +_l (o + 1\ \text{rem}\ cap) \leftarrow \mathsf{true}
\end{aligned}
$$

10.3 The specification

The specification can be seen as an extension of the specification for the ticket lock. The significant changes specific to the ABQL are highlighted in red in the presentation below.

$\exists\, \mathsf{isLock} : \mathsf{GhostName} \rightarrow \mathsf{GhostName} \rightarrow \mathsf{GhostName} \rightarrow Val \rightarrow \mathbb{N} \rightarrow \mathsf{Prop} \rightarrow \mathsf{Prop}.$

$\exists\, \mathsf{locked} : \mathsf{GhostName} \rightarrow \mathsf{GhostName} \rightarrow \mathsf{Prop}.$

$\exists\, \mathsf{issued} : \mathsf{GhostName} \rightarrow \mathbb{N} \rightarrow \mathsf{Prop}.$

$\exists\, \mathsf{invitation} : \mathsf{GhostName} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathsf{Prop}.$

$$
\begin{aligned}
& \forall \gamma, \iota, \kappa, l, cap, R.\ \mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R) \Rightarrow \square\, \mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R) \\
\wedge\ & \forall \iota, n, m, cap.\ \mathrm{invitation}(\iota, n, cap) * \mathrm{invitation}(\iota, n, cap) \Leftrightarrow \mathrm{invitation}(\iota, n + m, cap) \\
\wedge\ & \forall cap, R.\ \{R * 0 < cap\}\ \mathrm{newLock}\, cap\ \{l.\ \exists \gamma, \iota, \kappa.\ \mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R) * \mathrm{invitation}(\iota, cap, cap)\} \\
\wedge\ & \forall \gamma, \iota, \kappa, l, cap, R.\ \{\mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R) * \mathrm{invitation}(\iota, 1, cap)\}\ \mathrm{acquire}\ l\ \{o.\ \mathrm{locked}(\gamma, \kappa, o) * R\} \\
\wedge\ & \forall \gamma, \iota, \kappa, l, cap, R.\ \{\mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R) * \mathrm{issued}(\gamma, t)\}\ \mathrm{wait}\, l\ t\ \{\_.\ \mathrm{locked}(\gamma, \kappa, t) * R\} \\
\wedge\ & \forall \gamma, \iota, \kappa, l, cap, R.\ \{\mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R) * \mathrm{locked}(\gamma, \kappa, o) * R\}\ \mathrm{release}\ l\ o\ \{\_.\ \mathrm{invitation}(\iota, 1, cap)\}
\end{aligned}
$$

The $\mathrm{isLock}(\gamma, \iota, \kappa, l, cap, R)$ predicate represents the knowledge that the value l is a lock with a capacity of cap, and which protects the resource R. Since the lock is supposed to be used in a concurrent setting the *isLock* predicate is persistent, as we have seen before. The predicates $\mathrm{locked}(\gamma, \kappa, o)$ and $\mathrm{issued}(\gamma, t)$ serve the same purpose as they did in the ticket lock.

The precondition for *newLock* requires that a lock must be created with a capacity greater than zero. This is required as the implementation performs modulo with cap, which is not defined for zero.

The biggest change is the addition of *invitations*. Since every thread waiting to acquire the lock is spinning on an entry in the array, and since the array is cap long, at most cap threads may simultaneously attempt to acquire the lock.

The predicate $\mathrm{invitation}(\iota, n, cap)$ denotes the ownership of n invitations where cap invitations exists in total. When the lock is created, cap invitations are also constructed, see the postcondition for newLock. In order to acquire the lock, one invitation is required and is surrendered to the lock. When a thread later releases the lock, an invitation is given back, as seen in the postcondition for release, such that the thread can then acquire the lock again.

Since invitations are not duplicable, we ensure that at most cap threads can simultaneously attempt to acquire the lock. Invitations can be split and combined, meaning that the ownership of $n + m$ invitations implies the separate ownership of n and m invitations.

10.4 Representing invitations with ghost state

The invitation predicate is implemented as ghost state using the following resource algebra. We define the carrier of the resource algebra as

$$\mathcal{M} = (\mathbb{N} \times \mathbb{N}) \cup \bot.$$

The first number in the pair represents how many invitations are owned, and the second how many invitations exist in total.

It should not be possible to have more invitations than the total number of invitations, and hence the valid elements are defined as

$$\mathcal{V} = \{(n, m) \mid n \leq m\}.$$

Since it should be possible to combine and separate invitations, we define the operation as

$$(a,n)\cdot(b,m) = \begin{cases} (a+b,m) & \text{if } n = m \\ \bot & \text{otherwise} \end{cases}.$$

It only makes sense to combine invitations with the same upper bound, and hence the operation returns $\bot$ if the two upper bounds are not equal. Invitations should not be duplicable, and to this end the core function is always undefined.

With this resource algebra in place, we now define invitations as

$$\text{invitation}(\gamma,a,n) = \boxed{(a,n)}^{\gamma}.$$

Exercise 10.1. Show that the resource algebra defined above satisfies the requirements given in the definition of a resource algebra (Definition 7.10). $\diamond$

10.5 Implementation of the specification

Having understood the purpose of invitations we are now ready to show that the implementation meets the specification. We adopt the notation $\text{seq}(o,i)$ for the set of numbers $\{n \in \mathbb{N} \mid o \leq n < o+i\}$ and define the following predicates:

$$\begin{aligned}
\text{isLock}(\gamma,\iota,\kappa,l,cap,R) &= \exists a,n,\delta \in \varepsilon. \\
&\quad\; l = (a,n,cap) \;*\; 0 < cap \;* \\
&\quad\; \boxed{\text{lockInv}(\gamma,\iota,\kappa,a,cap,n,R)}^{\delta} \\
\text{lockInv}(\gamma,\iota,\kappa,a,cap,n,R) &= \exists o,i,xs. \\
&\quad\; n \hookrightarrow (o+i) \;*\; a \hookrightarrow_* xs * \text{length } xs = cap \;* \\
&\quad\; \text{invitation}(\iota,i,cap) * \boxed{\bullet(o,\text{seq}(o,i))}^{\gamma} * \\
&\quad\; \text{state}(\gamma,\kappa,cap,o,R,xs) \\
\text{state}(\gamma,\kappa,cap,o,R,xs) &= (\boxed{\circ(o,\emptyset)}^{\gamma} * R * \text{both}(\kappa) * xs = \text{nthTrue}(cap, o \text{ rem } cap)) \vee \\
&\quad\; (\text{issued}(\gamma,o) * \text{right}(\kappa) * xs = \text{replicate}(\mathsf{false}, cap)) \vee \\
&\quad\; (\text{issued}(\gamma,o) * \text{left}(\kappa) * xs = \text{nthTrue}(cap, o \text{ rem } cap)) \\
\text{issued}(\gamma,t) &= \boxed{\circ(\varepsilon,\{t\})}^{\gamma} \\
\text{locked}(\gamma,\kappa,o) &= \boxed{\circ(o,\emptyset)}^{\gamma} * \text{right}(\kappa)
\end{aligned}$$

In the following subsections we describe these predicates at a high-level and identify several useful and central properties as lemmas, which we then refer back to when we present the proofs. Hopefully this makes it easier for the reader to understand how the various pieces fit together.

10.5.1 The ghost state

The ghost state with ghost name γ is defined using the resource algebra $\text{Auth}(\text{Ex}(\mathbb{N})_? \times (\mathbb{P}_{fin}(\mathbb{N})))$. This is the same resource algebra used in the ticket lock. Intuitively, an element of this resource algebra is a pair where the first element can include knowledge about the value of o, which represents that the thread with this ticket currently has or may acquire the lock, and the second element knowledge about which tickets currently exist.

We use *disjoint* union such that partial knowledge about the existence of a ticket can only exist once. Hence we have the following lemma:

Lemma 10.2. *Only a single thread can know that a certain ticket has been issued.*

$$\boxed{\circ(\varepsilon, \{t\})}^{\gamma} * \boxed{\circ(\varepsilon, \{t\})}^{\gamma} \twoheadrightarrow \bot$$

Similarly, the exclusive construct around the value of o ensures that this information can also only exist once.

Lemma 10.3. *Only a single thread can have the partial information about o.*

$$\boxed{\circ(o, \emptyset)}^{\gamma} * \boxed{\circ(o, \emptyset)}^{\gamma} \twoheadrightarrow \bot$$

The predicates *left*(κ), *right*(κ), and *both*(κ) are defined using the resource algebra $\mathrm{Ex}(1)_? \times \mathrm{Ex}(1)_?$ where 1 denotes the unit resource algebra with the single element $()$:

$$\mathrm{left}(\kappa) = \boxed{((), \varepsilon)}^{\kappa} \qquad \mathrm{right}(\kappa) = \boxed{(\varepsilon, ())}^{\kappa} \qquad \mathrm{both}(\kappa) = \boxed{((), ())}^{\kappa}.$$

Lemma 10.4. *The resource both(κ) splits into left(κ) and right(κ), and left(κ) and right(κ) can be combined back together into both(κ).*

$$\begin{aligned} \mathit{both}(\kappa) &\twoheadrightarrow \mathit{left}(\kappa) * \mathit{right}(\kappa) \\ \mathit{left}(\kappa) * \mathit{right}(\kappa) &\twoheadrightarrow \mathit{both}(\kappa) \end{aligned}$$

Lemma 10.5. *Any other combination of left(κ), right(κ), and both(κ) except for those in Lemma 10.4 is invalid.*

Both lemmas follow directly from the definition of the resource algebra and the rules Own-op and Own-valid.

10.5.2 The *isLock* predicate

The *isLock* predicate states the existence of a, which is the array value, and n which is a location pointing to the index of the next ticket. The equality $l = (a, n, cap)$ describes the physical representation of the lock. The inequality $0 < cap$ serves to constrain cap such that we can use it to do division with remainder. Finally, *isLock* contains the invariant which we describe in the next subsection.

10.5.3 The invariant

The invariant *lockInv* states the existence of three things. A natural number o, representing that the thread with this ticket currently has or may acquire the lock. The number i, which represents how many threads are currently waiting to acquire the lock. Finally, *xs* is a list of booleans corresponding to the current contents of the array.

The invariant contains the points-to predicate for n (from the *isLock* predicate)

$$n \hookrightarrow o + i.$$

As one would expect, n points to the ticket currently granting access to the lock added together with the number of threads waiting for the lock.

Inside the invariant we have the authoritative part

$$\boxed{\bullet\,(o, \mathrm{seq}(o,i))}^{\gamma}.$$

The lock always knows the value of o and of all the tickets in existence. Here we have made an important change compared to the equivalent part in the ticket lock. In the ticket lock the set of tickets always starts from 0, and it is only expanded when new tickets are issued. In some sense, tickets are not "cleaned up" after their usage. For the ABQL it is important that the set of tickets starts at o. This means that whenever o increases, we know that all tickets smaller than o no longer exist. Hence, we have the following lemma.

Lemma 10.6. *From issued*(γ, o) *and the content of the invariant one can conclude* $o \leq t < o + i$. *More precisely,*

$$\boxed{\circ(\varepsilon, \{t\})}^{\gamma} * \boxed{\bullet\,(o, seq(o,i)}^{\gamma} \twoheadrightarrow \boxed{\circ(\varepsilon, \{t\})}^{\gamma} * \boxed{\bullet\,(o, seq(o,i)}^{\gamma} * o \leq t < o + i.$$

The lemma is straightforward to show. Suppose the two ghost states are owned. By Own-op and Own-valid the product of the two elements of the resource algebra is also owned and is valid. From the definition of the authoritative resource algebra and the disjoint set resource algebra this implies $\{t\} \subseteq \mathrm{seq}(o,i)$. Which per the definition of *seq* means that we have the bound in the lemma. This tighter bound is not needed for the proofs of the ticket lock to go through but for the ABQL it is necessary.

The invariant also contains ghost state for invitations, which we have already described. It is used as

$$\mathrm{invitation}(\iota, i, cap)$$

inside the invariant. This establishes the connection that i is the number of invitations owned by the lock. If i threads are currently waiting for the lock then i invitations are currently owned by the lock. This corresponds to the specification where, when calling *acquire*, a thread surrenders an invitation and when calling *release* the invitation is handed back. In the meantime the invitation is "stored" in the lock. As discussed previously the inclusion of invitations serves to constrain how many threads can wait for the lock. Indeed we have the following key property:

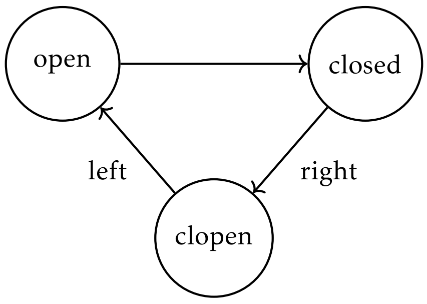
Lemma 10.7. *The* $\mathrm{invitation}(\iota, i, cap)$ *in the invariant establishes the inequality* $i \leq cap$.

$$\mathrm{invitation}(\iota, i, cap) \twoheadrightarrow i \leq cap$$

This lemma follows from the definition of invitation, Own-valid, and the definition of validity of the resource algebra. This lemma is essential when proving the specification for *wait*.

10.5.4 The *state* of the lock

The $\mathrm{state}(\gamma, \kappa, cap, o, R, xs)$ disjunction represents the current state of the lock. The first disjunct corresponds to the lock being *open* and the disjunct includes resource R since the lock owns the resource in this state. The last disjunct corresponds to the lock being *closed*. We call the middle disjunct *clopen*. It is necessary, since the release function proceeds in *two* atomic steps when it reopens the lock. Between these two steps the lock is neither closed nor open. Hence this disjunction is crucial in the proof of release.



In the first and third disjunct nthTrue(cap, o rem cap) denotes a mathematical sequence of booleans of length cap with false at every index except for index o rem cap which contains true.

10.5.5 The *issued* token

The issued(γ, t) token represents the information that the ticket t has been issued. The key property of this token is that if the resource issued(γ, o) is owned then it is possible to conclude that the lock is open.

Lemma 10.8. *Owning issued(γ, o) is sufficient to conclude that the lock is open.*

$$issued(\gamma, o) * state(\gamma, \kappa, cap, o, R, xs) \twoheadrightarrow (\boxed{\circ(o, \emptyset)}^{\gamma} * R * both(\kappa) * xs = nthTrue(cap, o \text{ rem } cap))$$

This follows since issued(γ, o) is incompatible with itself per Lemma 10.2 which leads to a contradiction in the *clopen* and *closed* branch of the *state* disjunction.

10.5.6 The *locked* predicate

The locked(γ, κ, o) predicate represents the knowledge that the lock is currently locked along with the knowledge of what o actually is. The ghost state right(κ) is included such that it is possible to conclude that the lock is in the *locked* state. This holds since the *open* state contains both(κ) and the *clopen* state contains right(κ), both of which contradicts with right(κ) per Lemma 10.5.

Lemma 10.9. *From locked(γ, κ, o) it is possible to conclude that the lock is closed.*

$$locked(\gamma, \kappa, o) * state(\gamma, \kappa, cap, o, R, xs) \twoheadrightarrow issued(\gamma, o) * left(\kappa) * xs = nthTrue(cap, o \text{ rem } cap)$$

10.6 Proofs

We now prove the specification for each of the functions.

10.6.1 Proof of newLock

We sketch the proof of *newLock* only at a high level as it is fairly trivial. After executing the alloc$_N$ and writing true at the first index with $(array +_l 0) \leftarrow$ true we know that the returned value represents an array with true at index zero and false at all other indices. We can then establish the *isLock* predicate by supplying a matching sequence of booleans as a witness. For n we supply the witness 0. We must then allocate an invariant and the ghost state it requires. Both of these can be done with the appropriate allocation rules and by framing. When establishing the *state* disjunction in the invariant, we show the disjunct corresponding to the lock being open.

10.6.2 Proof of acquire

For the *acquire* function we must prove the following specification:

$$\{\text{isLock}(\gamma,\iota,\kappa,l,cap,R) * \text{invitation}(\iota,1,cap)\}\ \text{acquire}\ \ l\,\{o.\ \text{locked}(\gamma,\kappa,o) * R\}$$

Recall that the function is defined by the following $\lambda_{\text{ref,conc}}$code.

```
let acquire l = let next = π₂ l in
                let ticket = faa next 1 in
                wait l ticket;
                ticket
```

From the *isLock* predicate we know that l is a triple and that there exists a location n which is the second element of the triple. Using these facts and structural rules we can step through the projection and the first let-expression. We now apply the bind rule on the expression faa n 1. Since faa is an atomic expression we can open the lock invariant. From opening the lock invariant we know that there exists natural numbers o and i such that n points to $o+i$. With this points-to predicate, we can apply the Ht-faa rule after which we know

$$n \hookrightarrow o+i+1$$

and that the faa n 1 expression evaluates to the value $o+i$.

We must now close the invariant, which per the definition means that we have to show:

$$\exists o,i,xs.\, n \hookrightarrow (o+i)\ * a \hookrightarrow_* xs * \text{length}\ xs = cap\ * \text{invitation}(\iota,i,cap) * \\ \boxed{\bullet(o,\text{seq}(o,i))}^{\gamma} * \text{state}(\gamma,\kappa,cap,o,R,xs)$$

The only part of the invariant we no longer have is $n \hookrightarrow o+i$. Instead, we now have $n \hookrightarrow o+i+1$. Thus, when we close the invariant we have to provide either $o+1$ as a witness for o or $i+1$ as a witness for i. Since o represents the index of who may currently access the lock and i represents how many threads are waiting, it only makes sense to increment i. Hence, we provide o as a witness for o and $i+1$ as a witness for i. For xs we provide the same value we got when opening the invariant. Since we only changed i we can frame away the part of the invariant that does not depend on i. The points-to assertion $n \hookrightarrow o+i+1$ can also be framed away since we picked i exactly such that it matched the points-to assertion we had after the fetch-and-add.

In order to close the invariant we are now left with showing

$$\text{invitation}(\iota,i+1,cap) * \boxed{\bullet(o,\text{seq}(o,i+1))}^{\gamma}.$$

From opening the invariant we have invitation(ι,i,cap) and from the precondition in the specification we have invitation$(\iota,1,cap)$. We can combine these two facts to get invitation$(\iota,i+1,cap)$ and frame the same thing away in the postcondition.

To show the second item we must update the authoritative ghost state we got when opening the invariant, namely $\bullet(o,\text{seq}(o,i))$. We can update it to $\bullet(o,\text{seq}(o,i)\cup\{o+i\})\cdot\circ(\varepsilon,\{o+i\})$. The first part of the product is exactly what we need to close the invariant, and the second is issued$(\varepsilon,o+i)$ which we will need later when calling *wait*.

After closing the invariant we can evaluate the let-expression and are then left with

$$\text{wait}\,l\ (o+i);o+i$$

Since the code is a sequencing of two expressions we apply the sequencing rule. We have the *isLock* predicate, $\text{issued}(\varepsilon, o{+}i)$, and the code $\text{wait}\, l\; (o{+}i)$. This is fits the wait specification, which we apply. The postcondition for the specification gives us the resource R and $\text{locked}(\gamma, \kappa, o+i)$. Additionally, the last expression evaluates to $o{+}i$. This matches precisely with the postcondition for the specification, and we are done by framing.

10.6.3 Proof of wait

We now prove the specification for the *wait* function.

$$\{\text{isLock}(\gamma, \iota, \kappa, l, cap, R) * \text{issued}(\gamma, t)\}\ \text{wait}\, l\ t\, \{\_.\ \text{locked}(\gamma, \kappa, t) * R\}.$$

Recall that the implementation of *wait* is.

```
let wait l t = let array = π₁ l in
               let i = t rem (π₃ l) in
               if !(array +ₗ i) then () else (wait l t)
```

The definition is recursive, so we apply the Ht-Rec rule, and assume that the specification holds for any recursive call.

From the *isLock* predicate we know that l is a triple and that there exists a value a, which is the first element of the triple, and a natural number cap, which is the third element of the triple. With this information we can step through the projections and the let-expressions. We then focus on the condition

$$!(array +_l i)$$

with the bind rule. We now open the invariant which contains the information that there exists an xs such that $array \hookrightarrow_* xs$. In order for the expression to make sense, the index must be smaller than the length of the array. Fortunately, we know from the invariant that the length of the array is cap and t rem cap is certain to be smaller than cap. From this we can show that the load expression evaluates to a boolean b where

$$b = xs_{(t\ \text{rem}\ cap)} \tag{38}$$

Since b is a boolean it is either true or false, and we proceed by case analysis on b.

We first consider the case where $b = \text{false}$ as it is the simplest case. It suffices to show:

```
if false then () else (wait l t)
```

We cannot step forward through the code until after we close the invariant. We still have everything from opening the invariant, so we simply use the exact same values as witnesses to close the invariant. We then apply Ht-If-False which leaves us with the code

$$\text{wait}\, l\ t$$

This is a recursive call of *wait* and we can now apply the specification we assumed when we used the Ht-Rec rule earlier. This concludes the first case.

In the second case we have $b = \text{true}$. Intuitively, we are in this case because the lock is open. Not only is the lock open, our ticket t should be equal to the ticket which now grants access to the lock, namely o. If we can show this, then we have $\text{issued}(\gamma, o)$, and we can then use Lemma 10.8 to conclude that the lock is open.

Consider the state($\gamma, \kappa, cap, o, R, xs$) part of the invariant:

$$\begin{aligned}
&(\boxed{\circ(o, \emptyset)}^{\gamma} * R * \mathrm{both}(\kappa) * xs = \mathrm{nthTrue}(cap, o \text{ rem } cap)) \vee \\
&(\mathrm{issued}(\gamma, o) * \mathrm{right}(\kappa) * xs = \mathrm{replicate}(\mathsf{false}, cap)) \vee \\
&(\mathrm{issued}(\gamma, o) * \mathrm{left}(\kappa) * xs = \mathrm{nthTrue}(cap, o \text{ rem } cap)).
\end{aligned}$$

The second case of the disjunction contains the equality

$$xs = \mathrm{replicate}(\mathsf{false}, cap)$$

This states that xs is a list containing only false. This is a contradiction, as we have just read true at an index in the array. In particular, we know that $b = \mathsf{true}$ and $b = xs_{(t \text{ rem } cap)}$. It is easy to show that $xs = \mathrm{replicate}(\mathsf{false}, cap)$ implies that $xs_i = \mathsf{false}$ for any i smaller than cap. And, since t rem cap is certainly smaller than cap we have the desired contradiction.

Both of the remaining cases contains the equality $xs = \mathrm{nthTrue}(cap, o \text{ rem } cap)$. Combining this with Equation 38 we get

$$\mathrm{nthTrue}(cap, o \text{ rem } cap)_{(t \text{ rem } cap)} = \mathsf{true}.$$

Recall that $\mathrm{nthTrue}(cap, o \text{ rem } cap)$ denotes a list that only contains true at index o rem cap. Thus the above implies

$$o \equiv t \pmod{cap}. \tag{39}$$

This is, unfortunately, not enough to prove $t = o$. It could, for instance, still be the case that $t = o - cap$ or $t = o + cap$. However, considering how the ABQL works neither of these can actually happen.

- We can never have a ticket that is smaller than o. Then it would already have been our turn to acquire the lock.
- Our ticket can not be larger than $o + cap$. Intuitively, $t - o$ represents how many threads are currently in front of us in the queue to acquire the lock. But, because at most cap threads can ever wait for the lock, our position in the queue can be no larger than cap.

Fortunately, both of these intuitions are encoded in the definitions and can be shown as follows.

We have $\mathrm{issued}(\gamma, t) = \boxed{\circ(\varepsilon, \{t\})}^{\gamma}$ and $\boxed{\bullet(o, \mathrm{seq}(o, i))}^{\gamma}$. By using Lemma 10.6 we have

$$o \leq t < o + i$$

This shows the first item above, that t can not be smaller than o.

From opening the invariant we have $\mathrm{invitation}(\iota, i, cap)$. We can thus use Lemma 10.7 to conclude that $i \leq cap$. Put together with the above, we have the second item.

$$t < o + i \leq o + cap.$$

This is sufficient to show that $t = o$. We leave the details as an exercise.

Exercise 10.10. Show that given $cap, t, o \in \mathbb{Z}$ where $0 < cap$ and $o \leq t < o + cap$ the equality t rem $cap = o$ rem cap implies that $t = o$. $\diamond$

This means that in both of the remaining cases we have $\mathrm{issued}(\gamma, o)$ and per Lemma 10.2 we now know that the lock must be in the state corresponding to open.

In other words, from opening the lock invariant we now also have both(κ), the resource R, and the partial information $\boxed{\circ(o, \emptyset)}^{\gamma}$. We can split both($\kappa$) into a left($\kappa$) and a right($\kappa$). This means that we have issued(γ, o), left(κ), and, from opening the invariant, $xs = \text{nthTrue}(cap, o \text{ rem } cap)$. These are exactly the things needed to show the *closed* state of the disjunction in the invariant. Thus we can close the invariant providing the same variables we initially got for o, i, and xs followed by framing.

We have now closed the invariant and can step further into the code. We use the Ht-If-True rule and are left with

$$()$$

Hence, the only thing left to show is the postcondition. This includes R and $\text{locked}(\gamma, \kappa, o) = \boxed{\circ(o, \emptyset)}^{\gamma} * \text{right}(\kappa)$. Both of these follow from framing as we have these from the invariant.

10.6.4 Proof of release

In this section we describe the proof of the specification for the *release* function. Recall the specification:

$$\{\text{isLock}(\gamma, \iota, \kappa, l, cap, R) * \text{locked}(\gamma, \kappa, o) * R\} \text{ release } l\ o \{\_.\ \text{invitation}(\iota, 1, cap)\}$$

Also recall that *release* is defined as follows.

```
let release l o = let array = π₁ l in
                  let cap = π₃ l in
                  array +ₗ (o ‘rem‘ cap) ← false;
                  array +ₗ (o + 1 ‘rem‘ cap) ← true
```

We can evaluate the two let expressions using the facts from the isLock predicate which leaves us with

```
array +ₗ (o ‘rem‘ cap) ← false;
array +ₗ (o + 1 ‘rem‘ cap) ← true
```

Before we proceed any further, let us take a step back and consider how the proof must proceed. In the code above, there are two store operations that write into the array. Since the points-to predicate for the array is inside the invariant we must open the invariant twice. Each time we open the invariant we must consider the state($\gamma, \kappa, cap, o, R, xs$) disjunction inside the invariant.

Since the lock is closed when we open the invariant for the first time, we should be able to conclude that it is in the *closed* state. We then set the single true value in the array to false. Hence, we must close the invariant in the *clopen* state. When we open the invariant for the second time, we should be able to conclude that the disjunction is still in the *clopen* state. Finally, after the last store operation we should be able to close the invariant in the *open* state.

We use the bind rule to focus on the first store operation.

```
array +ₗ (o ‘rem‘ cap) ← false
```

We now open the invariant and get the existence of o', i, and xs such that

$$n \hookrightarrow (o' + i) * array \hookrightarrow_* xs * \text{length } xs = cap *$$
$$\text{invitation}(\iota, i, cap) * \boxed{\bullet(o', \text{seq}(o', i))}^{\gamma} * \text{state}(\gamma, \kappa, cap, o', R, xs)$$

Since we have $\mathrm{right}(\kappa)$ we can conclude that the $\mathrm{state}(\gamma, \kappa, cap, o', R, xs)$ disjunction is in the last state as the first branch contains $\mathrm{both}(\kappa)$ and the second branch contains $\mathrm{right}(\kappa)$—both of which lead to a contradiction when combined with $\mathrm{right}(\kappa)$. We therefore additionally know.

$$\mathrm{issued}(\gamma, o') * \mathrm{left}(\kappa) * xs = \mathrm{nthTrue}(cap, o' \ \mathrm{rem}\ cap).$$

When closing the invariant we want to show the middle part of the state disjunction, and hence our goal is to show

$$\mathrm{issued}(\gamma, o') * \mathrm{right}(\kappa) * xs = \mathrm{replicate}(\mathsf{false}, cap).$$

The only thing we must work for is $xs = \mathrm{replicate}(\mathsf{false}, cap)$ as the other properties can be framed away. Fortunately, this is exactly what we expect to be able to show after stepping through the store as this operation is supposed to overwrite the single true in the array with a false. But, notice that we are setting the array using the value o, but, that the information from the invariant describes where the true value is in the list in terms of o'. We therefore want to show that o and o' are in fact equal. To do this we combine $\boxed{\circ(o, \emptyset)}^{\gamma}$ from the $\mathrm{locked}(\gamma, \kappa, o)$ predicate in the precondition with $\boxed{\bullet(o', \mathrm{seq}(o', i))}^{\gamma}$ from the invariant.

We now have $xs = \mathrm{nthTrue}(cap, o' \ \mathrm{rem}\ cap)$ and the code

$$array +_l (o \text{ `}rem\text{` } cap) \leftarrow \mathsf{false}$$

From this we can show the postcondition $xs = \mathrm{replicate}(\mathsf{false}, cap)$. This is what we need to close the invariant in the *clopen* state. We frame everything else away.

We now focus on the next store operation. We still have the resources we started with except that instead of $\mathrm{right}(\kappa)$ we now have $\mathrm{left}(\kappa)$, and the code is

$$array +_l (o + 1 \text{ `}rem\text{` } cap) \leftarrow \mathsf{true}$$

We again open the invariant. Similarly to how we previously used $\mathrm{right}(\kappa)$ to determine the state of the disjunction in the lock we now use $\mathrm{left}(\kappa)$ to determine that the lock is still in the clopen state. We therefore have $xs = \mathrm{replicate}(\mathsf{false}, cap)$ and $array \hookrightarrow \mathrm{replicate}(\mathsf{false}, cap)$. After executing the store operation we thus have

$$array \hookrightarrow \mathrm{nthTrue}(cap, (o+1)\ \mathrm{rem}\ cap)$$

We are now ready to close the invariant for the last time. When closing the invariant in the open state we must provide three witnesses and show the following.

$$\begin{aligned} \exists o, i, xs.\ & n \hookrightarrow (o + i) \ * a \hookrightarrow_* xs * \mathrm{length}\ xs = cap\ * \\ & \mathrm{invitation}(\iota, (i-1), cap) * \boxed{\bullet(o, \mathrm{seq}(o, i))}^{\gamma} * \\ & \boxed{\circ(o, \emptyset)}^{\gamma} * R * \mathrm{both}(\kappa) * xs = \mathrm{nthTrue}(cap, o\ \mathrm{rem}\ cap) \end{aligned}$$

For o we provide $o+1$, since we have now effectively moved the true value in the array one element to the left. For xs we provide $\mathrm{nthTrue}(cap, (o+1)\ \mathrm{rem}\ cap)$. When we opened the invariant every entry in xs was false but we have updated one entry to true. For i we provide $i-1$. We do this because invariant contains $n \hookrightarrow o + i$ and since we are providing $o+1$ in the place of o we have to establish $n \hookrightarrow (o+1)+i$. But, we have not changed what n points to so we still only have $n \hookrightarrow o + i$. The only way we can make this work is to use $i - 1$ as the witness for i.

With this choice of witnesses we can immediately frame away R and the points-to predicate for the array. The equality involving the length is trivially true. We show $\mathrm{both}(\kappa)$ by combining the $\mathrm{left}(\kappa)$ and the $\mathrm{right}(\kappa)$ that we have with Own-op followed by framing.

For the points-to predicate we have $n \hookrightarrow o + i$ and are to show $n \hookrightarrow o + 1 + (i - 1)$. Since we are working with natural numbers $(o + 1) + (i - 1)$ is only equal to $o + i$ if i is greater than 0. We thus have the points-to predicate if we can show that i is greater than 0.

From opening the invariant we have both $\text{issued}(\gamma, o) = \boxed{\circ(\varepsilon, \{o\})}^{\gamma}$ and $\boxed{\bullet(o, \text{seq}(o, i))}^{\gamma}$. We can combine these using Own-op and then conclude that

$$\circ(\varepsilon, \{o\}) \cdot \bullet(o, \text{seq}(o, i))$$

is valid from the Own-valid rule. From the definition of valid in the authoritative resource algebra, this implies that $o \in \text{seq}(o, i)$. If i was 0 then $\text{seq}(o, i)$ would be the empty set so this can not be the case. With this fact we can rewrite the goal $n \hookrightarrow ((o + 1) + (i - 1))$ into $n \hookrightarrow o + i$ which we can then frame away.

Notice that since we decremented the value of i when we closed the invariant we only have to show $\text{invitation}(\iota, i - 1, cap)$. On the other hand, the postcondition of *release* requires us to show $\text{invitation}(\iota, 1, cap)$. This adds up nicely. By using Own-op we split the i invitations we have into one ghost state with 1 invitation and one ghost state with $i - 1$ invitations. We can then frame both the aforementioned goals away.

The only thing that remains to show is:

$$\boxed{\bullet(o + 1, \text{seq}(o + 1, i - 1))}^{\gamma} * \boxed{\circ(o + 1, \emptyset)}^{\gamma}$$

To do that we have $\boxed{\circ(\varepsilon, \{o\})}^{\gamma}$ and $\boxed{\bullet(o, \text{seq}(o, i))}^{\gamma}$ from opening the invariant and $\boxed{\circ(o, \emptyset)}^{\gamma}$ from the $\text{locked}(\gamma, \kappa, o)$ predicate in the original precondition. It is clear that we must make a frame preserving update in order to achieve this. Specifically we need the frame preserving update

$$\circ(o, \emptyset) \cdot \circ(\varepsilon, \{o\}) \cdot \bullet(o, \text{seq}(o, i)\})) \rightsquigarrow \circ(o + 1, \emptyset) \cdot \bullet(o + 1, \text{seq}(o + 1, i - 1))).$$

Exercise 10.11. Show the above frame preserving update. ◊

Using the Ghost-update rule on this frame preserving update we are done with the proof of *release*.

10.7 Discussion

We have now completed the proof of the specification for the ABQL, a lock that can be used by at most a fixed number of participating threads. In particular, we have seen how the specification of the ticket lock can be extended to express the restrictions using the concept of invitations and we have verified that the implementation meets the specification using a suitable resource algebra for invitations.

11 Modular Specifications for Concurrent Modules

In the previous sections we have seen several examples and case studies involving specifications of concurrent modules. In particular, in Section 7.7 we presented several different specifications of a simple counter module. In general, it is difficult to find out what is the "right" specification to give to a (concurrent) module. Often we would like to have a specification which is sufficiently general that it can be used by many, ideally *all possible*, different clients. In this section we give some "methodological advice" on how to give modular specifications for concurrent modules that are sufficiently general that they can be instantiated by many diverse clients. In particular, we present a new specification of the concurrent counter module, which is *more modular*, in the sense that the earlier given specifications can be derived from it (*without reference to the code of the counter, only using the abstract predicates*). Moreover, we also show how this modular specification of the counter module allows us to give a modular proof of the ticket lock, *i.e.*, a proof of the ticket lock which only depends on the *specification* of the counter module, *not* on the concrete implementation. We include this section for the obvious reason that modularity is a key point we have been striving for all along, but also because it gives us an additional opportunity to show how Iris's higher-order logic supports quite advanced modular specifications.

The methodology we present is a *higher-order approach to modular specifications of concurrent modules*. It stems from [14], which was based on [5]. It is closely related to the notion of logical atomicity from the TaDa logic [1]. The examples, the concurrent counter module and the ticket lock, are from [2], which contains a presentation and discussion of these examples using TaDa-style logical atomicity. This section is supposed to be an introduction to the topic of modular specifications for concurrent modules, please see the mentioned references for further discussion.

We now outline the overall idea of the methodology; it is perhaps a little tricky to understand at first, so it may be easiest to read this description quickly at first, and then study the examples below and return to the description again.

We consider concurrent modules which have some state and some methods operating on the state. A concrete example could be a concurrent stack module. To specify a concurrent module, we decide on what the mathematical model of the abstract state should be, and in particular how the model allows for sharing. For the concurrent stack module, a natural choice is to model the abstract state of the stack by a mathematical sequence of natural numbers (assuming that the elements of the stack are simply natural numbers). We will use a ghost variable to keep track of the contents of the abstract state of the module, so for the concurrent stack we will have a ghost resource whose contents will be a mathematical sequence of numbers. Now consider the specification of a method. Typically, it will involve a modification of the abstract state of the module. For example, for the concurrent stack, a push method is supposed to change the abstract state of the module, by inserting the element being pushed into the front of the mathematical sequence modeling the abstract state of the stack.

Since we are in a concurrent setting, it matters *when* the state of the module changes, and when the abstract state changes, a client will typically also have to update some invariants and protocols of its own. However, the module, of course, cannot know how different clients wish to update their invariants when the abstract state of the module changes. Therefore we *parameterize* the method specification by a view shift, which (1) describes how the abstract state is supposed to change and (2) describes how other invariants should be updated. The idea is that, when we prove the specification of the method of the module, then we can use this view shift to update the abstract state of the module; typically, we will then also show that the concrete state of the module matches the new abstract state of the module. Thus it is the *client* of the module who has to prove that the abstract state of the module can be changed as described by the view shift, since the client has to provide a proof of the view shift. (Perhaps it is surprising that the *client* can prove that the abstract state of the module can be changed, but notice that the client only considers the *abstract state* of the module, which is tracked using ghost state — the modifications to the actual *concrete state* of the module are proved to match the abstract state change when we prove the module method specification.) Since the module cannot know which other invariants the client has, we also parameterize the specification by predicates intended to describe those.

11.1 Modular Specification of Concurrent Counter Module

Counter Implementation The counter implementation we will consider is the same as in Section 7.7, except we add an additional *weak increment* (wk_incr) method. Its definition is the following

$$\text{wk\_incr}\, \ell = \ell \leftarrow 1 + !\ell.$$

The intention is that clients should only call the weak increment method when the client knows that it is safe to do so, *i.e.*, when the client knows that only one thread will increment the counter. Since the increment is not atomic

A Resource Algebra for the Abstract State of the Counter We model the abstract state of the counter by a natural number. We will use a resource algebra for keeping track of the abstract state of the counter module. The resource algebra is the product of the resource algebra of fractions (from Example 7.17) and the agreement resource algebra (from Example 7.14) on the set of natural numbers $\mathbb{N}$ (the type of the model of the abstract state of the counter). We write $\gamma \Mapsto^q m$ for the ghost ownership assertion $\boxed{(q,m)}^{\gamma}$ to reflect the intuitive reading of this ghost ownership assertion as "there is a ghost heap, which maps γ to m with fraction q". In particular, we write $\gamma \Mapsto_{\frac{1}{k}} m$ when the fraction q is $\frac{1}{k}$. It is a simple exercise to verify that for all n, m and p, q we have the following entailments.

$$\gamma \Mapsto^q m * \gamma \Mapsto^q n \vdash n = m \tag{40}$$

$$\gamma \Mapsto^p m * \gamma \Mapsto^q m \dashv\vdash \gamma \Mapsto^{p+q} m \tag{41}$$

$$\gamma \Mapsto^1 m \vdash \mid\!\Rrightarrow \gamma \Mapsto^1 n \tag{42}$$

The first property means that everybody in possession of the partial knowledge (fraction q less than 1) agrees on the value of the counter, the second property states how the abstract predicate can be split, and the third property states that anybody in full possession of the abstract state of the counter can update it.

Exercise 11.1. Prove the preceding three properties of the assertion $\gamma \Mapsto^q m$. ◊

Modular Counter Specification In the following we assume $\mathcal{E}$ is an infinite set of invariant names.

The modular counter specification is as follows, we explain it below. In the specification for wk_incr, P and Q range over Prop, v over Val, q over fractions $\mathbb{Q}_{01}$, and m over $\mathbb{N}$.

$$
\begin{aligned}
&\exists \mathsf{Cnt} : \mathsf{Val} \to \mathsf{GhostName} \to \mathsf{InvName} \to \mathsf{Prop}.\\
&\quad \Box(\forall v, \gamma, c.\ \mathsf{Cnt}(v,\gamma,c) \Rightarrow \Box\mathsf{Cnt}(v,\gamma,c))\\
&\wedge \quad \{\mathsf{True}\}\ \text{newCounter}()\ \left\{v.\exists \gamma, c.\ \mathsf{Cnt}(v,\gamma,c) * \gamma \Mapsto_{\frac{1}{2}} 0\right\}_{\mathcal{E}}\\
&\wedge \quad \forall \gamma, c, P, Q, v.\ \left(\forall m. (\gamma \Mapsto_{\frac{1}{2}} m * P) \Rrightarrow_{\mathcal{E}\setminus\{c\}} (\gamma \Mapsto_{\frac{1}{2}} m * Q(m))\right) \Rightarrow\\
&\qquad\qquad\qquad\qquad \{\mathsf{Cnt}(v,\gamma,c) * P\}\ \text{read}(v)\ \{u. \mathsf{Cnt}(v,\gamma,c) * Q(u)\}_{\mathcal{E}}\\
&\wedge \quad \forall \gamma, c, P, Q, v.\ \left(\forall m. (\gamma \Mapsto_{\frac{1}{2}} m * P) \Rrightarrow_{\mathcal{E}\setminus\{c\}} \left(\gamma \Mapsto_{\frac{1}{2}} (m+1) * Q(m)\right)\right) \Rightarrow\\
&\qquad\qquad\qquad\qquad \{\mathsf{Cnt}(v,\gamma,c) * P\}\ \text{incr}(v)\ \{u. \mathsf{Cnt}(v,\gamma,c) * Q(u)\}_{\mathcal{E}}\\
&\wedge \quad \forall \gamma, c, P, Q, v, q, m.\ \left(\gamma \Mapsto_{\frac{1}{2}} m * \gamma \Mapsto^q m * P \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma \Mapsto_{\frac{1}{2}} (m+1) * \gamma \Mapsto^q (m+1) * Q\right) \Rightarrow\\
&\qquad\qquad\qquad\qquad\qquad \{\mathsf{Cnt}(v,\gamma,c) * \gamma \Mapsto^q m * P\}\ \text{wk\_incr}(v)\ \{u. u = () * \mathsf{Cnt}(v,\gamma,c) * Q\}_{\mathcal{E}}
\end{aligned}
$$

The idea is that the abstract predicate $\mathrm{Cnt}(v,\gamma,c)$ expresses that v represents a counter, whose abstract state is kept in the ghost variable γ, and which uses invariant name c. As usual, $\mathrm{Cnt}(v,\gamma,c)$ is persistent so that we can share it among several threads.

The postcondition of newCounter says that a counter is created and, moreover, that the abstract state of the counter is 0. The client of the counter gets fractional $(\frac{1}{2})$ ownership of the abstract state, which means that if the client gets access to the other remaining fraction $(\frac{1}{2})$, which is kept by the counter module, then it can update the abstract state of the counter. This also means that the counter module cannot update the abstract state of the counter "on its own" (remember that the idea of the methodology is that the module cannot know what should happen when the abstract state changes and hence it delegates updating of the abstract state to the client of the module).

Now consider the specification of incr. To use this specification, the client must show the view shift

$$\forall m. (\gamma \Mapsto^{\frac{1}{2}} m * P) \Rrightarrow_{\mathcal{E}\setminus\{c\}} (\gamma \Mapsto^{\frac{1}{2}} (m+1) * Q(m)),$$

and then it gets the Hoare triple $\{\mathrm{Cnt}(v,\gamma,c) * P\}\,\mathrm{incr}(v)\,\{u. \mathrm{Cnt}(v,\gamma,c) * Q(u)\}_{\mathcal{E}}$. The view shift expresses that the incr will increment the abstract state of the counter; the predicates P and Q are universally quantified and can thus be instantiated by the client to coordinate updates to invariants held by the client. The Hoare triple expresses that one may call incr if one has a $\mathrm{Cnt}(v,\gamma,c)$ resource.

The specification of read is similar to the specification of incr, except that the abstract state does not change. Even though the abstract state does not change, we still parameterize the specification by a view shift, because that will allow a client to update its own invariants appropriately when it learns about the abstract state of the counter (by instantiating P and Q as necessary). We will show examples of how this can be done below.

Note that the quantification over the abstract state m of the counter in the view shifts for incr and read captures the point that a client cannot know (if the counter is shared by different threads) what the abstract state is — because other threads may call methods on the counter concurrently.

In the specification of wk_incr, the value of the counter m is quantified over both the view shift and the Hoare triple. Moreover, to call wk_incr, the client must have fragmental ownership of the abstract state of the counter (note the $\gamma \Mapsto^{q} m$ in the precondition of the Hoare triple); this captures the idea that no other thread can have full ownership of the abstract state and hence cannot update the abstract state "under our feet", which is in accordance with the idea that a client should only call wk_incr when it knows that no other thread can modify the counter. In the specifications for incr and read, the predicate Q is parameterized by the abstract state of the counter (because we do not know up front what the abstract state is), but in the specification for wk_incr, the predicate Q need not be parameterized by the abstract state of the counter, since the client already keeps track of it $(\gamma \Mapsto^{q} m)$.

Finally, we comment on the mask annotation on the view shifts: since the mask is $\mathcal{E}\setminus\{c\}$, the client may use (open and close) all the invariants in $\mathcal{E}$ when showing the view shift, except the invariant named c used by the counter module. That is also the reason why we parameterize Cnt by c (rather than hiding c behind an existential, as we have done in earlier examples). (In Coq, we use invariant name spaces to keep tract of these invariant names, see Section 12.5.)

Showing that the Implementation meets the Modular Counter Specification We now outline the proof that the counter implementation meets the above modular specification. We naturally use an invariant to share the state of the counter. The invariant connects the concrete value of the counter to the abstract state of the counter, which in this case is simply the same value as the concrete value stored in the reference of the counter module. [15]

$$\begin{aligned}\mathrm{CntInv}(v,\gamma) &= \exists m.\, v \hookrightarrow m * \gamma \Mapsto_{\frac{1}{2}} m\\ \mathrm{Cnt}(v,\gamma,c) &= \boxed{\mathrm{CntInv}(v,\gamma)}^{c}\end{aligned}$$

With this definition of the abstract Cnt predicate, it is not hard to show that the different methods meet the specifications. Here we just outline the proof for the incr method, and leave the other (easier) ones as an exercise.

For incr we first assume the given view shift, and the proceed to show the Hoare triple. To that end, since incr is a recursive function we proceed, as usual, by Löb induction. To dereference the reference we open the invariant and then close it again. Then we get to the cas instruction. We open the invariant and thus get fractional ownership of the abstract state, *i.e.*, we get $\gamma \Mapsto_{\frac{1}{2}} m$ for some m. The interesting case is when it succeeds (otherwise we just end up recursing so the proof succeeds by applying the Löb induction hypothesis). In this case we get that the reference now points to $m+1$ (since the cas succeeded). Now we want to apply the view shift, so we instantiate it with m, and then we can apply it. This we can do since we both have $\gamma \Mapsto_{\frac{1}{2}} m$ and P (we have P from the precondition in the Hoare triple). By the view shift we have $\gamma \Mapsto_{\frac{1}{2}} (m+1)$ and $Q(m)$. Thus, since we now both have that the reference points to $m+1$ and we also have $\gamma \Mapsto_{\frac{1}{2}} (m+1)$, we can close the counter invariant, and thus we obtain the required postcondition. So, in summary, the key point to notice is that the abstract state of the counter is updated by an application of the view shift which the specification is parameterized by.

Exercise 11.2. Show the specifications for newCounter, read, and wk_incr. ◊

11.1.1 Deriving Counter with Contributions from the Modular Counter Specification

In this subsection we sketch how we may use the modular counter specification from above to *derive* a counter-with-contributions specification from Exercise 7.50 in Section 7.7.

The idea is to proceed much as in Exercise 7.50, except that now we have to use the *abstract state* of the counter (as a client of the modular counter specification that is all we can use!). Thus we let isCounter be the predicate

$$\mathrm{isCounter}(\ell,n,\gamma_1,\gamma_2,c,p) = \boxed{\circ(p,n)}^{\gamma_1} * \exists \iota \in \mathcal{E}\setminus\{c\}.\, \boxed{\exists m.\, \gamma_2 \Mapsto_{\frac{1}{2}} m * \boxed{\bullet(1,m)}^{\gamma_1}}^{\iota} * \mathrm{Cnt}(\ell,\gamma_2,c).$$

where we use the same authoritative resource algebra as we did for the verification of the counter with contributions previously. Note the similarity to the earlier definition in Exercise 7.50! In the definition of isCounter, the predicate $\mathrm{Cnt}(\ell,\gamma_2,c)$ expresses that ℓ is a counter, whose abstract state is tracked by γ_2, and in the invariant we use $\gamma_2 \Mapsto_{\frac{1}{2}} m$ to record that the abstract state of the counter is m (note how this ghost state plays a role similar to the role played by $\ell \hookrightarrow m$ in Exercise 7.50). Also note that $\mathrm{isCounter}(\ell,n,\gamma_1,\gamma_2,c,p)$ is persistent.

With this definition in place, we can prove the following specifications:

$$\begin{aligned}&\{\mathrm{True}\}\ \mathrm{newCounter}()\ \{u.\exists \gamma_1,\gamma_2,c.\ \mathrm{isCounter}(u,0,\gamma_1,\gamma_2,c,1)\}\\ &\forall p.\forall \gamma_1,\gamma_2.\forall c.\forall v.\forall n.\ \{\mathrm{isCounter}(v,n,\gamma_1,\gamma_2,c,p)\}\ \mathrm{read}\, v\ \{u.u \geq n * \mathrm{isCounter}(v,n,\gamma_1,\gamma_2,c,p)\}\\ &\forall \gamma_1,\gamma_2.\forall c.\forall v.\forall n.\ \{\mathrm{isCounter}(v,n,\gamma_1,\gamma_2,c,1)\}\ \mathrm{read}\, v\ \{u.u = n * \mathrm{isCounter}(v,n,\gamma_1,\gamma_2,c,1)\}\\ &\forall p.\forall \gamma_1,\gamma_2.\forall c.\forall v.\forall n.\ \{\mathrm{isCounter}(v,n,\gamma_1,\gamma_2,c,p)\}\ \mathrm{incr}\, v\ \{u.u = () * \mathrm{isCounter}(v,n+1,\gamma_1,\gamma_2,c,p)\}\end{aligned}$$

[15] Generally, in this methodology, the abstract state is an appropriate mathematical abstraction of the contents of the module, *e.g.*, the abstract state for a concurrent stack module could be a mathematical sequence of values.

We sketch the proof for incr, and the leave the others as an exercise. We assume the precondition, which gives us $\mathrm{Cnt}(v,\gamma_2,c)$, as is necessary for using the modular counter specification for incr. We instantiate P and Q in the modular incr specification by $P = \boxed{\circ(p,n)}^{\gamma_1}$ and $Q = \lambda x.\boxed{\circ(p,n+1)}^{\gamma_1}$. Now we need to show the view shift

$$\gamma_2 \Mapsto_{\frac{1}{2}} m * \boxed{\circ(p,n)}^{\gamma_1} \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma_2 \Mapsto_{\frac{1}{2}} (m+1) * \boxed{\circ(p,n+1)}^{\gamma_1}.$$

We do this by opening the invariant ι, which gives us $\gamma_2 \Mapsto_{\frac{1}{2}} k * \boxed{\bullet(1,k)}^{\gamma_1}$, for some k. By the properties for resource algebra for abstract state (40) we conclude that $k = m$ and that $\gamma_2 \Mapsto^{1} k$. By combining the fraction of the abstract state from the invariant with the fraction from the assumption in the view shift, using (41) and (42), we can then update the abstract state to $\gamma_2 \Mapsto^{1} (k+1)$, and by the properties of the authoritative resource algebra we can also update $\boxed{\circ(p,n)}^{\gamma_1} * \boxed{\bullet(1,k)}^{\gamma_1}$ to $\boxed{\circ(p,n+1)}^{\gamma_1} * \boxed{\bullet(1,k+1)}^{\gamma_1}$. With this we can close the ι invariant again. Recalling that $k = m$, the resources we have left are exactly the $\gamma_2 \Mapsto_{\frac{1}{2}} (m+1) * \boxed{\circ(p,n+1)}^{\gamma_1}$, as required for completing the proof of the view shift.

Since we have shown the view shift, we now get the Hoare triple

$$\left\{\mathrm{Cnt}(v,\gamma_2,c) * \boxed{\circ(p,n)}^{\gamma_1}\right\} \mathrm{incr}(v) \left\{u.u = () * \mathrm{Cnt}(v,\gamma_2,c) * \boxed{\circ(p,n+1)}^{\gamma_1}\right\}$$

By the definition of isCounter, we not only have $\mathrm{Cnt}(v,\gamma_2,c)$, but also $\boxed{\circ(p,n)}^{\gamma_1}$. Hence we conclude from the Hoare triple above that we can indeed call $\mathrm{incr}(v)$ and obtain $\mathrm{Cnt}(v,\gamma_2,c) * \boxed{\circ(p,n+1)}^{\gamma_1}$, which, together with the invariant ι, suffices to conclude $\mathrm{isCounter}(v,n{+}1,\gamma_1,\gamma_2,c,p)$, as required.

11.1.2 Deriving Sequential Counter from Modular Counter Specification

Here is a specification for a counter that can be used in a sequential context only:

$$\begin{aligned}
&\exists\, \mathsf{SeqCnt} : \mathsf{Val} \to \mathsf{GhostName} \to \mathsf{InvName} \to \mathbb{N} \to \mathsf{Prop}.\\
&\qquad \forall n.\, \{\mathsf{True}\}\ \mathrm{newCounter}()\ \{v.\exists \gamma, c.\ \mathrm{SeqCnt}(v,\gamma,c,0)\}\\
&\wedge \quad \forall \gamma, c, v, n.\ \ \{\mathrm{SeqCnt}(v,\gamma,c,n)\}\ \mathrm{read}(v)\ \{u.u = n * \mathrm{SeqCnt}(v,\gamma,c,n)\}\\
&\wedge \quad \forall \gamma, c, v, n.\ \ \{\mathrm{SeqCnt}(v,\gamma,c,n)\}\ \mathrm{incr}(v)\ \{u.u = n * \mathrm{SeqCnt}(v,\gamma,c,n+1)\}
\end{aligned}$$

Since the representation predicate $\mathrm{SeqCnt}(v,\gamma,c,n)$ is *not* persistent, it cannot be duplicated, which means that we can only use it sequentially. On the other hand, because we can only use the counter sequentially, we can track the precise value of the counter (not just a lower bound).

We can easily derive this specification from the modular counter specification by defining $\mathrm{SeqCnt}(v,\gamma,c,n) = \mathrm{Cnt}(v,\gamma,c) * \gamma \Mapsto_{\frac{1}{2}} n$. Then, for instance, to derive the specification for incr, we let $P = \gamma \Mapsto_{\frac{1}{2}} n$ and $Q = \lambda n.\gamma \Mapsto_{\frac{1}{2}} (n+1)$ in the modular counter specification for incr. Note how we track the precise value of the abstract state using $\gamma \Mapsto_{\frac{1}{2}} n$. and that, indeed, with this definition, $\mathrm{SeqCnt}(v,\gamma,c,n)$ is not persistent.

11.2 Modular Verification of the Ticket Lock

In this section we give a modular proof of a modular implementation of the ticket lock from Section 9. Recall that the earlier verified version of the ticket lock was not modular in its use of counters; that is what we change now. Thus the ticket lock implementation we wish to verify now is:

$$\begin{aligned}
\texttt{let}\,\mathrm{newLock}() &= (\mathrm{newCounter}(), \mathrm{newCounter}()) \\
\texttt{let}\,\mathrm{acquire}\,l &= \texttt{let}\,n = \pi_2\,l\,\texttt{in} \\
&\ \mathrm{wait}(\mathrm{incr}\,n)l \\
\texttt{let}\,\mathrm{wait}\,n\,l &= \texttt{let}\,o = \mathrm{read}(\pi_1\,l)\,\texttt{in} \\
&\ \texttt{if}\,n = o\,\texttt{then}\,()\,\texttt{else}\,\mathrm{wait}\,n\,l \\
\texttt{let}\,\mathrm{release}\,l &= \mathrm{wk\_incr}(\pi_1\,l)
\end{aligned}$$

Note the use of the modular counter methods, the calls to: newCounter in newLock, incr in acquire, read in wait, and wk_incr in release.

We want to give this version of the ticket lock almost the same specification as before.

$$\begin{aligned}
&\exists\,\mathrm{isLock} : \mathsf{Val} \to \mathsf{Prop} \to \mathsf{GhostName} \to \mathsf{GhostName} \to \mathsf{GhostName} \to \mathsf{InvName} \to \mathsf{InvName} \to \mathsf{Prop}. \\
&\exists\,\mathrm{locked} : \mathsf{GhostName} \to \mathsf{GhostName} \to \mathsf{Prop}. \\
&\exists\,\mathrm{issued} : \mathsf{GhostName} \to \mathsf{Val} \to \mathsf{Prop}. \\
&\quad\ \forall P, v, \gamma, \gamma_o, \gamma_n, c_o, c_n.\ \mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n) \Rightarrow \square\mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n) \\
&\wedge\ \ \forall \gamma, \gamma_o.\ \mathrm{locked}(\gamma, \gamma_o) * \mathrm{locked}(\gamma, \gamma_o) \Rightarrow \mathsf{False} \\
&\wedge\ \ \forall \gamma.\ \mathrm{issued}(\gamma, n) * \mathrm{issued}(\gamma, n) \Rightarrow \mathsf{False} \\
&\wedge\ \ \forall P.\ \{P\}\ \mathrm{newLock}()\ \{v.\exists \gamma, \gamma_o, \gamma_n, c_o, c_n.\ \mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n)\} \\
&\wedge\ \ \forall P, v, \gamma, \gamma_o, \gamma_n, c_o, c_n, n.\ \{\mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n) * \mathrm{issued}(\gamma, n)\}\ \mathrm{wait}(n, v)\ \{v.P * \mathrm{locked}(\gamma, \gamma_o)\} \\
&\wedge\ \ \forall P, v, \gamma, \gamma_o, \gamma_n, c_o, c_n.\ \{\mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n)\}\ \mathrm{acquire}(v)\ \{v.P * \mathrm{locked}(\gamma, \gamma_o)\} \\
&\wedge\ \ \forall P, v, \gamma, \gamma_o, \gamma_n, c_o, c_n.\ \{\mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n) * P * \mathrm{locked}(\gamma, \gamma_o)\}\ \mathrm{release}(v)\ \{\_.\mathsf{True}\}
\end{aligned}$$

As you can see, the only change (compared to the earlier specification in Section 9) is the additional ghost name and invariant name arguments to the abstract predicates and invariant. That is because we use the modular counter specification, which is also parameterized by ghost names and invariant names.

To prove that the implementation above meets the specification, we define the abstract predicates as follows:

$$\begin{aligned}
\mathrm{lockInv}(\gamma, \gamma_o, \gamma_n, P) = \exists o, n.\ &\gamma_o \Mapsto_{\frac{1}{4}} o * \gamma_n \Mapsto_{\frac{1}{2}} n * \boxed{\bullet\,(o, \{i \mid 0 \le i < n\})}^{\gamma} \\
&* ((\boxed{\circ\,(o, \emptyset)}^{\gamma} * \gamma_o \Mapsto_{\frac{1}{4}} o * P) \vee \boxed{\circ\,(\varepsilon, \{o\})}^{\gamma}).
\end{aligned}$$

$$\begin{aligned}
\mathrm{isLock}(v, P, \gamma, \gamma_o, \gamma_n, c_o, c_n) &= \exists \ell_o, \ell_n, \iota \in \mathsf{InvName}.\ v = (\ell_o, \ell_n) * \boxed{\mathrm{lockInv}(\gamma, \gamma_o, \gamma_n, P)}^{\iota} \\
&\qquad * \mathsf{Cnt}(\ell_o, \gamma_o, c_o) * \mathsf{Cnt}(\ell_n, \gamma_n, c_n) \\
\mathrm{locked}(\gamma, \gamma_o, \gamma_n) &= \exists o. \boxed{\circ\,(o, \emptyset)}^{\gamma} * \gamma_o \Mapsto_{\frac{1}{4}} o \\
\mathrm{issued}(\gamma, n) &= \boxed{\circ\,(\varepsilon, \{n\})}^{\gamma}
\end{aligned}$$

Compared to the earlier non-modular proof of the ticket lock, the change is that our definitions are now given in terms of the abstract modular counter predicates (rather than relying on information about the actual implementation of counters).

As in the derivation of the counter with contributions from the modular counter, we use abstract state predicates $\gamma_o \Mapsto_{\frac{1}{4}} o$ and $\gamma_n \Mapsto_{\frac{1}{2}} n$ to track the values of the owner and the next counter. The reason for using different fractions for the owner counter and the next counter is that we want to be able to call wk_incr on the owner counter when releasing the lock and the specification of wk_incr requires some fragmental ownership of the corresponding ghost state in its precondition, and hence we split the half ownership of γ_o into two quarters, one of which we keep in the "basic" part of the invariant and the other of which we keep in the left side of the disjunction and in the locked predicate. Remember how this left side corresponds to the lock not being in use at the moment, so having this ghost resource in both the invariant and the predicate makes sense, since only one of these will ever be true.

11.2.1 wait-loop

The wait method now calls read on the owner counter instead of loading its contents directly, so we need to utilise the counter module's read specification now. Apart from this, the proof follows the same structure as before.

Recall that the intuition in the entire method is that we have a ticket with a specific number and read the owner counter until its value matches the one on our ticket. When these two values match, we "hand in" our ticket and actually acquire the lock, which means that we need to change our state. As long as they do not match, we start over and do not change any state. This intuition corresponds to the instantiation of P and Q in the counter module's read specification:

$$\begin{aligned} P &= \text{issued}(\gamma, n) \\ Q &= \lambda v.(\text{locked}(\gamma, \gamma_o, \gamma_n) * P \vee \text{issued}(\gamma, n)) \end{aligned}$$

Note that Q is the same as the one we chose as intermediate postcondition for our bind rule in the old proof.

This means we need to show the view shift:

$$\forall m.\, \gamma_o \mapsto_{\frac{1}{2}} m * \text{issued}(\gamma_o, n) \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma_o \mapsto_{\frac{1}{2}} m * (\text{locked}(\gamma) * P \vee \text{issued}(\gamma_o, n)).$$

Opening the invariant gives us $\gamma_o \mapsto_{\frac{1}{2}} o$ for some o. As in the preceding section, we then conclude that $o = m$. As in the old proof, we proceed by casing on whether $n = o$. If they are not the same, we choose to show the right side of the disjunction in Q, which we can do by simply closing the invariant again. If they are the same, however, we need to show that we can update our abstract state to locked. This is accomplished the same way as in the old proof, i.e., by concluding that the left side of the disjunction in the invariant must have been true before and then closing the invariant again with our ticket instead, thereby releasing exactly the resources corresponding to locked and the lock resources needed for the postcondition of our specification.

11.2.2 acquire

The proof of the acquire specification is even closer to the old one than wait. The cas operation is now replaced by the incr method, so we make use of its specification instead. We instantiate $P = \text{True}$ and $Q = \lambda v.\, \text{issued}(\gamma_n, v)$ in the counter module's incr specification and have to prove the view shift

$$\forall m.\, \gamma_n \mapsto_{\frac{1}{2}} m * \text{True} \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma_n \mapsto_{\frac{1}{2}} (m+1) * \text{issued}(\gamma_n, m).$$

Opening the invariant gives us the other half permission for γ_n containing some m' and, as earlier, we can conclude that $m = m'$. Moreover, we get the authoritative part of the ticket lock resources and can update them as we did in the earlier proof, leaving us with the issued(γ_n, m), as needed for the postcondition of the view shift. With this ticket we can now use our wait specification from above and are done with the proof.

11.2.3 release

Release makes use of the wk_incr method, so we need to be able to provide some fragmental ownership of γ_o. This is contained in the locked predicate, which is part of our precondition. Intuitively, it makes sense to use wk_incr rather than incr, since only the thread currently holding the lock is allowed to call release. This is reflected in the formal specifications in the way that the only way to get the locked predicate is to have succeeded with acquire.

We instantiate P by $\boxed{\circ(o,\emptyset)}^{\gamma} * R$ and Q by True in the counter module's wk_incr specification. The R in P are the resources the lock protects and we therefore currently own, since we have the lock. At this point, we have to prove the view shift

$$\gamma_o \Mapsto_{\frac{1}{2}} m * \gamma_o \Mapsto_{\frac{1}{4}} m * \boxed{\circ(m,\emptyset)}^{\gamma} * R \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma_o \Mapsto_{\frac{1}{2}} (m+1) * \gamma_o \Mapsto_{\frac{1}{4}} (m+1) * \mathsf{True}.$$

Note how m is not universally quantified in this specification. This corresponds to the fact that we know that no other threads can change the value of the owner counter since we have our fragmental ownership of the corresponding ghost resource. Opening the invariant will give us the last quarter ownership of γ_o. Again, we conclude the value it holds must be the same as m, so we can update γ_o to contain $m+1$. For the ticket lock ghost state we proceed as we did before in order to update it. We can then close the invariant with the resources for the left side of the disjunction and are done.

11.3 Summary

Above we have presented a higher-order approach to modular specifications of concurrent modules, where method specifications are parameterized by view shifts expressing (1) how the abstract state of the module changes by calling the method and (2) how client invariants should be updated when the abstract state of the module changes. We have exemplified the method by presenting modular specifications of counters and shown how they can be used to verify modular implementations of ticket locks. In the accompanying Coq examples, you can find more examples, including a modular specification of the ticket lock and the concurrent bag and concurrent runner examples from [14].

As mentioned in the introduction to this chapter, our methodology for modular specifications of concurrent modules is related to the notion of logical atomicity from the TaDa logic. Indeed, the examples we have presented thus far can also be specified and verified using the Iris formalization of TaDa-style logical atomicity from [8].

The original presentation of the TaDa approach focuses on atomicity and was aimed at giving logically atomic specifications to methods that, to a client, appear to be atomic. The higher-order approach we have presented here focuses on changes to the abstract state and thus also applies to operations that are not logically atomic. For example, consider the following operation:

$$\text{incr\_twice}(\ell) = \text{incr}\,\ell;;\text{incr}\,\ell$$

This method does not have a single linearization point, so the fact that we cannot give it a logically atomic specification should not be a surprise. We can, however, use our higher-order approach if we use two view shifts, one for each modification of the abstract state. The specification looks as follows:

$$\begin{aligned}\forall \gamma, P, Q,' Q, \ell.\ & \big(\forall n.\, \gamma \Mapsto_{\frac{1}{2}} n * P \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma \Mapsto_{\frac{1}{2}} (n+1) * Q'(n)\big) \Rightarrow \\ & \quad \big(\forall n.\, \gamma \Mapsto_{\frac{1}{2}} n * (\exists m.\, Q'(m)) \Rrightarrow_{\mathcal{E}\setminus\{c\}} \gamma \Mapsto_{\frac{1}{2}} (n+1) * Q(n)\big) \Rightarrow \\ & \quad \{\mathsf{Cnt}(\ell, \gamma, c) * P\}\ \mathsf{incr\_twice}(\ell)\ \{r.\, \mathsf{Cnt}(\ell, \gamma, c) * Q(r)\}\end{aligned}$$

This said, one can use a modification of the Iris formalization of TaDa-style logically atomic triples to give a specification of incr_twice. Indeed, the two approaches are essentially equivalent.

12 First steps towards the base logic

The logic introduced thus far is powerful and can be used to verify 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", reducing the number of primitives of the logic, and defining as much as possible inside the logic itself. That is, of course, not only important for understanding and semantic modelling, 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 13).

12.1 Weakest precondition

We start off by reducing the notion of a Hoare triple to 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.

$$\frac{\mathcal{E} \subseteq \mathsf{InvName} \qquad \Gamma \vdash e : \mathit{Exp} \qquad \Gamma \vdash \Phi : \mathsf{Val} \to \mathsf{Prop}}{\Gamma \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} : \mathsf{Prop}}$$

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 $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.Q\}$. As evident in the typing rule, in the assertion $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ e is a closed term, Φ is an assertion, and $\mathcal{E}$ is the set of invariant names, playing the same role as it does in Hoare triples.

Remark 12.1. One can think of the mapping $\Phi \mapsto \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ as the semantics of the term capturing those aspects of the behaviour we care about, e.g., safety. In this way it embeds the terms of the programming language into assertions of the logic. ■

The intended meaning of the weakest precondition becomes clearer when we define Hoare triples in terms of it as

$$\{P\}\, e\, \{\Phi\}_{\mathcal{E}} \triangleq \Box(P \twoheadrightarrow \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}).$$

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 $\square$ modality is crucial. Indeed, without it the Hoare triple assertion would not be duplicable, which would mean that we could not use the specification of the method we proved more than once. Using the $\square$ modality guarantees that all the non-persistent resources required by e are contained in P, *i.e.*, that all exclusive resources e needs to run safely are 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 122. 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 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 12.2. The rule

$$\textsc{wp-load-direct}\qquad \frac{}{\triangleright(\ell \hookrightarrow v) \vdash \mathsf{wp}\ !\ell\,\{u.u = v * \ell \hookrightarrow v\}}$$

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 12.3. 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.

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq \exists P.\, P * \{P\}\, e\, \{\Phi\}_{\mathcal{E}}.$$

Structural rules.

WP-MONO
$$\frac{}{(\forall v.\, \Phi(v) \mathrel{-\!\!*} \Psi(v)) * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Psi\}}$$

WP-FRAME
$$\frac{}{P * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{P * \Phi\}}$$

WP-FRAME-STEP
$$\frac{e \notin \mathsf{Val}}{\triangleright P * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{P * \Phi\}}$$

WP-VAL
$$\frac{}{\Phi(v) \vdash \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\}}$$

WP-BIND
$$\frac{}{\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \mathsf{wp}_{\mathcal{E}}\, E[\, v\,]\, \{\Phi\}\} \vdash \mathsf{wp}_{\mathcal{E}}\, E[\, e\,]\, \{\Phi\}}$$

Rules for basic language constructs.

WP-FORK
$$\frac{}{\triangleright \Phi() * \triangleright \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \mathsf{True}\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{fork}\ \{e\}\ \{\Phi\}}$$

WP-ALLOC
$$\frac{}{\triangleright(\forall \ell.\, \ell \hookrightarrow v \mathrel{-\!\!*} \Phi(\ell)) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{ref}(v)\, \{\Phi\}}$$

WP-LOAD
$$\frac{}{\triangleright(\ell \hookrightarrow v) * \triangleright(\ell \hookrightarrow v \mathrel{-\!\!*} \Phi(v)) \vdash \mathsf{wp}_{\mathcal{E}}\, !\ell\, \{\Phi\}}$$

WP-STORE
$$\frac{}{\triangleright(\ell \hookrightarrow v) * \triangleright(\ell \hookrightarrow w \mathrel{-\!\!*} \Phi()) \vdash \mathsf{wp}_{\mathcal{E}}\, (\ell \leftarrow w)\, \{\Phi\}}$$

WP-CAS-SUC
$$\frac{}{\triangleright(\ell \hookrightarrow v) * \triangleright(\ell \hookrightarrow w \mathrel{-\!\!*} \Phi(\mathsf{true})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{cas}(\ell, v, w)\, \{\Phi\}}$$

WP-CAS-FAIL
$$\frac{}{v \neq v' \wedge \triangleright(\ell \hookrightarrow v) * \triangleright(\ell \hookrightarrow v \mathrel{-\!\!*} \Phi(\mathsf{false})) \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{cas}(\ell, v', w)\, \{\Phi\}}$$

WP-REC
$$\frac{}{\triangleright \mathsf{wp}_{\mathcal{E}}\, e[v/x][(\mathsf{rec}\, f(x) = e)/f]\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, (\mathsf{rec}\, f(x) = e) v\, \{\Phi\}}$$

WP-PROJ
$$\frac{}{\triangleright \mathsf{wp}_{\mathcal{E}}\, v_i\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \pi_i\, (v_1, v_2)\, \{\Phi\}}$$

WP-IF-TRUE
$$\frac{}{\triangleright \mathsf{wp}_{\mathcal{E}}\, e_1\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{if}\, \mathsf{true}\, \mathsf{then}\, e_1\ \mathsf{else}\, e_2\, \{\Phi\}}$$

WP-IF-FALSE
$$\frac{}{\triangleright \mathsf{wp}_{\mathcal{E}}\, e_2\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{if}\, \mathsf{false}\, \mathsf{then}\, e_1\ \mathsf{else}\, e_2\, \{\Phi\}}$$

WP-MATCH
$$\frac{}{\triangleright \mathsf{wp}_{\mathcal{E}}\, e_i\, [u/x_i]\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{match}\, \mathsf{inj}_i\, u\, \mathsf{with}\, \mathsf{inj}_1\, x_1 \Rightarrow e_1 \mid \mathsf{inj}_2\, x_2 \Rightarrow e_2\, \mathsf{end}\, \{\Phi\}}$$

Figure 10: Rules for the weakest precondition assertion.

- 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*[16], so it should not be tied to Hoare triples or the weakest precondition assertion directly. To support it we introduce a new concept, the *fancy update modality*.

12.2 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.

[16]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.

$$\textsc{Fup-mono}\quad \frac{P \vdash Q}{{}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} Q} \qquad \textsc{Fup-intro-mask}\quad \frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_1} P} \qquad \textsc{Fup-trans}\quad \frac{}{{}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_3} P}$$

$$\textsc{Fup-frame}\quad \frac{\mathcal{E}_f \text{ disjoint from } \mathcal{E}_1 \cup \mathcal{E}_2}{Q * {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}{\mid\!\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P)} \qquad \textsc{Fup-upd}\quad \frac{}{\mid\!\Rrightarrow P \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} P} \qquad \textsc{Fup-timeless}\quad \frac{\vdash P \text{ timeless}}{\triangleright P \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} P} \qquad \textsc{Inv-alloc}\quad \frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash {}^{\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_2} \exists \iota \in \mathcal{E}_1. \boxed{P}^{\iota}}$$

$$\textsc{Inv-open}\quad \frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright P * \left(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}{\mid\!\Rrightarrow}^{\mathcal{E}} \mathsf{True}\right)\right)}$$

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.

$$\textsc{Fup-mono}\quad \frac{P \vdash Q}{{}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} Q} \qquad \textsc{Fup-intro-mask}\quad \frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_1} P} \qquad \textsc{Fup-trans}\quad \frac{}{{}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}\, {}^{\mathcal{E}_2}{\mid\!\Rrightarrow}^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_3} P}$$

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 $P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P$. Indeed, if, for example, $P \vdash {}^{\emptyset}{\mid\!\Rrightarrow}^{\{\iota\}} P$ 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:

$$\textsc{Fup-intro}\quad \frac{}{P \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} P}$$

We will write ${\mid\!\Rrightarrow}_{\mathcal{E}} P$ for ${}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} P$.

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 $\mathcal{E}_f$.

$$\textsc{Fup-frame}\quad \frac{\mathcal{E}_f \text{ disjoint from } \mathcal{E}_1 \cup \mathcal{E}_2}{Q * {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}{\mid\!\Rrightarrow}^{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P)}$$

The rule perhaps 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.

$$\frac{}{Q * {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} (Q * P)} \qquad \frac{\mathcal{E}_1 \subseteq \mathcal{E}_2}{{\mid\!\Rrightarrow}_{\mathcal{E}_1} P \vdash {\mid\!\Rrightarrow}_{\mathcal{E}_2} P}$$

Exercise 12.4. 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
$$\frac{}{\Rrightarrow P \vdash \Rrightarrow_{\mathcal{E}} P}$$

Note that in combination with the previous rules for ${}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2}$, the rules Ghost-alloc and Ghost-update remain valid if we replace $\Rrightarrow$ with $\Rrightarrow_{\mathcal{E}}$, for any mask $\mathcal{E}$.

Fancy update modality and invariants Finally, we have rules for allocation and opening of invariants:

Inv-alloc
$$\frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_2} \exists \iota \in \mathcal{E}_1. \boxed{P}^{\iota}}$$

Inv-open
$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright P * \left(\triangleright P -\!\!* {}^{\mathcal{E}\setminus\{\iota\}}\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right)}$$

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
$$\frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash {}^{\emptyset}\Rrightarrow^{\emptyset} \exists \iota \in \mathcal{E}_1. \boxed{P}^{\iota}}$$

Exercise 12.5. 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

$$\frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \triangleright P} \qquad \frac{\iota \in \mathcal{E}}{\boxed{P}^{\iota} \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\{\iota\}} \left(\triangleright P -\!\!* {}^{\mathcal{E}\setminus\{\iota\}}\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)}$$

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. 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 12.6. Show the following property of the fancy update modality.

$${}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} (P -\!\!* Q) \vdash P -\!\!* {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} Q$$

◇

12.3 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
$$\frac{}{\models\!\!\!\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{v. \models\!\!\!\Rrightarrow_{\mathcal{E}} \Phi(v)\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

wp-atomic
$$\frac{e \text{ is an atomic expression}}{{}^{\mathcal{E}_1}\!\models\!\!\!\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \{v. {}^{\mathcal{E}_2}\!\models\!\!\!\Rrightarrow^{\mathcal{E}_1} \Phi(v)\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

wp-frame-step
$$\frac{e \notin \mathsf{Val} \qquad \mathcal{E}_2 \subseteq \mathcal{E}_1}{\left({}^{\mathcal{E}_1}\!\models\!\!\!\Rrightarrow^{\mathcal{E}_2} \triangleright {}^{\mathcal{E}_2}\!\models\!\!\!\Rrightarrow^{\mathcal{E}_1} P\right) * \mathsf{wp}_{\mathcal{E}_2}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{P * \Phi\}}$$

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 $\models\!\!\!\Rrightarrow_{\mathcal{E}} P \vdash P$, and so proving $\models\!\!\!\Rrightarrow_{\mathcal{E}} P$ 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 12.7. 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 7.6, 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 12.8. 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.

wp-inv-open
$$\frac{e \text{ is an atomic expression}}{\boxed{I}^{\iota} * \left(\triangleright I -\!\!* \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

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) \\
&\hfill (\textsc{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) \\
&\hfill (\textsc{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) \qquad (\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))\} \quad (\textsc{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)\} \qquad (\textsc{wp-mono} \text{ 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)\} \qquad (\textsc{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)\} \qquad (\textsc{Fup-mono}) \\
&\vdash \mathsf{wp}\, e\, \{\Phi\} \qquad (\textsc{wp-atomic})
\end{aligned}$$

■

The rule derived in the preceding example can be strengthened somewhat.

Exercise 12.9. Derive the following rules.

$$\left({}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \boxed{I}^{\iota}\right) * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \tag{43}$$

$${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \left(\boxed{I}^{\iota} * \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right)\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \tag{44}$$

$$\boxed{I}^{\iota} * {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \tag{45}$$

$$\boxed{I}^{\iota} * \left(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\{\iota\}}{\Rrightarrow}^{\mathcal{E}\setminus\{\iota\}}\, \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \tag{46}$$

These rules perhaps look rather strange, however they show that as long as we are proving a weakest precondition we can most of the time strengthen the assumptions by removing the fancy update modalities, provided the masks match. The rules are thus quite crucial in concrete proofs, and are often used implicitly. In particular when Iris is used in Coq via the interactive proof mode (see Section 13) these, and related rules, are used by the tactics behind the scenes.

◊

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 12.8, as we now demonstrate.

Example 12.10 (Derivation of Ht-inv-open from wp-inv-open)**.** Let e be an atomic expression. We are to show

$$\textsc{Ht-inv-open}\qquad \frac{S \wedge \boxed{I}^{\iota} \vdash \{\triangleright I * P\}\, e\, \{v. \triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}}}{S \wedge \boxed{I}^{\iota} \vdash \{P\}\, e\, \{\Phi\}_{\mathcal{E}}}$$

recalling that we have defined $\{P\}\, e\, \{\Phi\}_{\mathcal{E}}$ as $\square(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\})$. Let us show it. Since invariants and Hoare triples are persistent we have

$$\begin{aligned}
S \wedge \boxed{I}^{\iota} &\vdash (S \wedge \boxed{I}^{\iota}) \wedge \boxed{I}^{\iota} \\
&\vdash (\{\triangleright I * P\}\, e\, \{v. \triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}}) \wedge \boxed{I}^{\iota} \\
&\vdash \square\left(\{\triangleright I * P\}\, e\, \{v. \triangleright I * \Phi(v)\}_{\mathcal{E}\setminus\{\iota\}} * \boxed{I}^{\iota}\right)
\end{aligned}$$

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 PERSISTENTLY-MONO. In fact by PERSISTENTLY-E it suffices to show

$$\left(\triangleright I * P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) * \boxed{I}^{\iota} \vdash P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

which is equivalent to showing

$$\left(\triangleright I * P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) * \boxed{I}^{\iota} * P \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

by the wand introduction rule. Now

$$\left(\triangleright I * P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) * \boxed{I}^{\iota} * P \vdash \left(\triangleright I \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) * \boxed{I}^{\iota}$$

by the wand elimination rule, which in turn yields

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

by WP-INV-OPEN derived in Example 12.8. ■

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 12.11. Derive the following rules from WP-FRAME-STEP.

$$\frac{e \notin \mathrm{Val}}{\triangleright P * \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{v. P * \Phi(v)\}} \qquad \frac{e \notin \mathrm{Val} \qquad S \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}}}{S \vdash \{P * \triangleright R\}\, e\, \{v.Q * R\}_{\mathcal{E}}}$$

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

$$P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \triangleq \square(P \mathrel{-\!\!*} {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q).$$

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 12.12. Derive the following rules for the fancy view shift.

- Fvs-refl

  $$\frac{}{\cdot \vdash P \Rrightarrow_{\mathcal{E}_1} P}$$

  Fvs-trans

  $$\frac{S \vdash P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \qquad S \vdash Q \;{}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_3} R}{S \vdash P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_3} R}$$

- Fvs-imp

  $$\frac{S \vdash \square(P \Rightarrow Q)}{S \vdash P \Rrightarrow_{\mathcal{E}} Q}$$

  Fvs-wand

  $$\frac{S \vdash \square(P \mathrel{-\!\!*} Q)}{S \vdash P \Rrightarrow_{\mathcal{E}} Q}$$

- Fvs-frame

  $$\frac{S \vdash P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q}{S \vdash P * R \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q * R}$$

  Fvs-mask-frame

  $$\frac{S \vdash P \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} Q \qquad (\mathcal{E}_1 \cup \mathcal{E}_2) \cap \mathcal{E}_f = \emptyset}{S \vdash P * R \;{}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rrightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} Q * R}$$

- Fvs-timeless

  $$\frac{\vdash P \text{ timeless}}{\cdot \vdash \triangleright P \Rrightarrow_{\mathcal{E}} P}$$

- Fvs-alloc-I

  $$\frac{\mathcal{E} \text{ infinite}}{\cdot \vdash \triangleright P \;{}^{\emptyset}\!\Rrightarrow^{\emptyset} \exists \iota \in \mathcal{E}. \boxed{P}^{\iota}}$$

  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

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 12.13. Derive the above rule of consequence. ◊

The next rule is a generalisation of Ht-frame-atomic.

Ht-frame-step

$$\frac{e \notin \mathsf{Val} \qquad S \vdash \{P\} e \{v. Q\}_{\mathcal{E}_2} \qquad S \vdash R_1 \;{}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2} \triangleright R_2 \qquad S \vdash R_2 \;{}^{\mathcal{E}_2}\!\Rrightarrow^{\mathcal{E}_1} R_3 \qquad \mathcal{E}_2 \subseteq \mathcal{E}_1}{S \vdash \{P * R_1\} e \{v. Q * R_3\}_{\mathcal{E}_1}}$$

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 12.14. Derive the rule Ht-frame-step from the rule wp-frame-step. ◊

Example 12.15 (Improved Specfication for the Spin Lock). In this example we will show how to use the fancy update modality to give a better specification of the spin lock module from Section 7.6.

First, recall the specification we gave earlier for the spin lock module:

$$\begin{aligned}
&\exists\, \mathsf{isLock} : \mathsf{Val} \to \mathsf{Prop} \to \mathsf{GhostName} \to \mathsf{Prop}.\\
&\exists\, \mathsf{locked} : \mathsf{GhostName} \to \mathsf{Prop}.\\
&\qquad \square(\forall P, v, \gamma.\ \mathrm{isLock}(v,P,\gamma) \Rightarrow \square \mathrm{isLock}(v,P,\gamma))\\
&\wedge \quad \forall \gamma.\ \mathrm{locked}(\gamma) * \mathrm{locked}(\gamma) \Rightarrow \mathsf{False}\\
&\wedge \quad \forall P.\ \{P\}\ \mathrm{newLock}()\ \{v.\exists \gamma.\ \mathrm{isLock}(v,P,\gamma)\}\\
&\wedge \quad \forall P, v, \gamma.\ \{\mathrm{isLock}(v,P,\gamma)\}\ \mathrm{acquire}\, v\ \{\_.P * \mathrm{locked}(\gamma)\}\\
&\wedge \quad \forall P, v, \gamma.\ \{\mathrm{isLock}(v,P,\gamma) * P * \mathrm{locked}(\gamma)\}\ \mathrm{release}\, v\ \{\_.\mathsf{True}\}
\end{aligned}$$

Notice that the resource invariant, the predicate P, is in the precondition for the newLock method. This means that that a client of the lock module must allocate the resources, which the lock is going to protect, *before* calling the newLock method. For example, we cannot use the above specification to verify safety of the following simple client, where the intention, of course, is that the lock should protect the reference r.

$$C_1 \equiv \mathsf{let}\, l = \mathrm{newLock}()\, \mathsf{in}\, \mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\ \mathrm{acquire}\, l; r \leftarrow\, !r + 1; \mathrm{release}\, l$$

The problem is that when newLock is called, reference r is not in scope and thus we cannot instantiate P with our intended resource invariant $\exists n.r \hookrightarrow n$ when attempting to verify the call to newLock. Thus with the specification given above, we are forced to rewrite the client code as follows:

$$C_2 = \mathsf{let}\, r = \mathsf{ref}(0)\, \mathsf{in}\, \mathsf{let}\, l = \mathrm{newLock}()\, \mathsf{in}\ \mathrm{acquire}\, l; r \leftarrow\, !r + 1; \mathrm{release}\, l$$

so that the resource (here the reference r) is allocated before newLock is called. In general this is undesirable since the two programs are completely equivalent, and thus the logic should not force us to make trivial changes to the program in order to verify them.

Let us instead consider the following specification of the newLock method:

$$\{\mathsf{True}\}\ \mathrm{newLock}()\ \{v.\exists \gamma.\forall P.\ P \twoheadrightarrow {\Rrightarrow_{\emptyset}}\ \mathrm{isLock}(v,P,\gamma)\} \tag{47}$$

This specification expresses that we can always call newLock (since the precondition is True) and, moreover, that when newLock returns, we get the assertion

$$\forall P.\ P \twoheadrightarrow {\Rrightarrow_{\emptyset}}\ \mathrm{isLock}(v,P,\gamma). \tag{48}$$

The idea is that we can use this assertion when we know what the resource invariant P should be instantiated with. Then, when we have the resources P, we can use the assertion (48) to obtain ${\Rrightarrow_{\emptyset}}\ \mathrm{isLock}(v,P,\gamma)$, which is equivalent to $\mathrm{isLock}(v,P,\gamma)$ if it appears in the pre- or post-condition of a Hoare triple, *i.e.*, the following inference rules are valid.

$$\frac{\{P\}\, e\, \{v.Q\}_{\mathcal{E}}}{\{{\Rrightarrow_{\emptyset}}\, P\}\, e\, \{v.Q\}_{\mathcal{E}}} \qquad\qquad \frac{\{P\}\, e\, \{v.{\Rrightarrow_{\emptyset}}\, Q\}_{\mathcal{E}}}{\{P\}\, e\, \{v.Q\}_{\mathcal{E}}}$$

Exercise 12.16. Derive the preceding two rules from the rule of consequence. ◊

Remark 12.17. Notice that the $P \twoheadrightarrow \Rrightarrow_{\emptyset} \mathrm{isLock}(v, P, \gamma)$ is almost a fancy view shift, except for the missing $\square$ modality. The specification with a fancy view shift, *i.e.*,

$$\{\mathsf{True}\}\ \mathrm{newLock}()\ \{v.\exists\gamma.\forall P.\ \square(P \twoheadrightarrow \Rrightarrow_{\emptyset} \mathrm{isLock}(v, P, \gamma))\} \tag{49}$$

would be *unsound* in the sense that $\mathrm{isLock}(v, P, \gamma)$ would not protect the resources as explained in the following exercise. It would allow us to conjure up many different independent locks protecting the same resource, meaning none of the locks would actually protect the resource. ■

Exercise 12.18. Let e be the following program.

$$\begin{aligned}
&\mathsf{let}\, v = \mathrm{newLock}()\, \mathsf{in}\\
&\mathsf{let}\, \ell = \mathsf{ref}(0)\, \mathsf{in}\\
&\ \mathrm{acquire}\, v;\\
&\ \mathrm{release}\, v;\\
&\ \mathrm{acquire}\, v;\\
&\mathsf{let}\, n = !\,\ell\, \mathsf{in}\ \mathrm{release}\, v; n
\end{aligned}$$

Assuming (49) as the specification for newLock show the following specification.

$$\{\mathsf{True}\}\, e\, \{v.v = 37\}. \qquad \diamond$$

The specifications of the acquire and release methods stay the same.

Exercise 12.19.

1. Use the new lock module specification to verify the safety of the first client program, by showing the following specification:

$$\{\mathsf{True}\}\, C_1\, \{\mathsf{True}\}$$

2. Prove that the newLock method for the spin lock implementation in Section 7.6 satisfies the specification in (47). ◊

■

12.4 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

$$\textsc{Fup-timeless}\quad \frac{\vdash P\ \mathsf{timeless}}{\triangleright P \vdash {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P}$$

which 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. The technical definition is as follows.

Definition 12.20. A proposition P is timeless if the following entailment holds

$$\triangleright P \vdash P \vee \triangleright \mathsf{False}$$

We write

$$\vdash P \ \mathsf{timeless}$$

for the judgement stating that P is timeless, or

$$\Gamma \vdash P \ \mathsf{timeless}$$

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.

$$\frac{}{\vdash \mathsf{True}\ \mathsf{timeless}} \qquad \frac{}{\vdash \mathsf{False}\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \vee Q\ \mathsf{timeless}}$$

$$\frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \wedge Q\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless} \qquad \vdash Q\ \mathsf{timeless}}{\vdash P \Rightarrow Q\ \mathsf{timeless}}$$

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

$$\frac{\Gamma, x : \tau \vdash \Phi\ \mathsf{timeless}}{\Gamma \vdash \exists x. \Phi\ \mathsf{timeless}} \qquad \frac{\vdash P\ \mathsf{timeless}}{\vdash \square P\ \mathsf{timeless}}$$

Moreover ghost ownership is timeless.

GHOST-RA-TIMELESS
$$\frac{M \text{ is a resource algebra} \qquad a \in M}{\vdash \boxed{a}^{\gamma}\ \mathsf{timeless}}$$

Figure 13: Rules for timeless propositions.

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.

$$\textsc{Ht-timeless-pre-post}\quad \frac{\vdash P_1 \text{ timeless} \qquad \vdash Q_1 \text{ timeless} \qquad \{P_1 * P_2\}\, e\, \{v. \triangleright Q_1 * Q_2\}_{\mathcal{E}}}{\{\triangleright P_1 * P_2\}\, e\, \{v. Q_1 * Q_2\}_{\mathcal{E}}}$$

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 12.21. Derive the rule Ht-timeless-pre-post from the generalized rule of consequence involving the fancy view shifts introduced in the previous section. $\diamond$

Exercise 12.22. Derive the following rules for timeless propositions.

$$\textsc{Ht-timeless-pre}\quad \frac{\vdash P_1 \text{ timeless} \qquad \{P_1 * P_2\}\, e\, \{v. Q\}_{\mathcal{E}}}{\{\triangleright P_1 * P_2\}\, e\, \{v. Q\}_{\mathcal{E}}} \qquad\qquad \textsc{Ht-timeless-post}\quad \frac{\vdash Q_1 \text{ timeless} \qquad \{P\}\, e\, \{v. \triangleright Q_1 * Q_2\}_{\mathcal{E}}}{\{P\}\, e\, \{v. Q_1 * Q_2\}_{\mathcal{E}}}$$

$\diamond$

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 $\triangleright\left[\bar{a}\right]^{\gamma}$ 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 12.23. Derive the following rules for invariant opening. We assume $\mathcal{E}$ is a set of invariant names and $\iota \in \mathcal{E}$.

$$\frac{\begin{array}{lll} e \text{ is an atomic expression} & \vdash I \text{ timeless} & \iota \in \mathcal{E} \end{array}}{\boxed{I}^{\iota} * \left(I \twoheadrightarrow \mathsf{wp}_{\mathcal{E}\setminus\{\iota\}}\, e\, \{v.I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\;\textsc{wp-inv-timeless}$$

$$\frac{\begin{array}{llll} e \text{ is an atomic expression} & \vdash I \text{ timeless} & \iota \in \mathcal{E} & S \wedge \boxed{I}^{\iota} \vdash \{I * P\}\, e\, \{v.I * Q\}_{\mathcal{E}\setminus\{\iota\}} \end{array}}{S \wedge \boxed{I}^{\iota} \vdash \{P\}\, e\, \{v.Q\}_{\mathcal{E}}}\;\textsc{Ht-inv-timeless}$$

◇

The exercise establishes some properties of timeless assertions which are used implicitly when the Iris logic is used in the Coq proof assistant.

Exercise 12.24. Assuming P is timeless derive the following rules.

$$\frac{Q * P \vdash \Rrightarrow_{\mathcal{E}} R}{Q * \triangleright P \vdash \Rrightarrow_{\mathcal{E}} R} \qquad \frac{Q * P \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}{Q * \triangleright P \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

◇

12.5 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

$$\frac{\mathcal{E}_1 \text{ infinite}}{\triangleright P \vdash \Rrightarrow_{\emptyset} \exists \iota \in \mathcal{E}_1. \boxed{P}^{\iota}}\;\textsc{Inv-alloc}$$

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 $(\mathcal{N}.\text{lock})^{\uparrow}$ and $(\mathcal{N}.\text{counter})^{\uparrow}$ are disjoint. To make use of namespaces we define some abbreviations. We define[17]

$$\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}^{\uparrow}. \boxed{P}^{\iota}.$$

With this notation the invariant allocation rule looks simpler, as

$$\triangleright P \vdash \Rrightarrow_{\emptyset} \boxed{P}^{\mathcal{N}}.$$

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

$$\frac{\mathcal{N}^{\uparrow} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}^{\uparrow}} \left(\triangleright P * \left(\triangleright P \twoheadrightarrow {}^{\mathcal{E}\setminus\mathcal{N}^{\uparrow}}\Rrightarrow^{\mathcal{E}} \mathsf{True}\right)\right)}\ \textsc{Inv-open-namespace}$$

and the rule for opening invariants in connection with the weakest precondition assertion is thus

$$\frac{e \text{ is an atomic expression} \qquad \mathcal{N}^{\uparrow} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} * \left(\triangleright I \twoheadrightarrow \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}^{\uparrow}}\, e\, \{v. \triangleright I * \Phi(v)\}\right) \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}\ \textsc{wp-inv-open-namespace}$$

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, 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 12.25. 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 $\mathcal{E}_f$.

$${}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \left(P * (Q \twoheadrightarrow {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_1} R)\right) \vdash {}^{\mathcal{E}_1 \cup \mathcal{E}_f}\Rrightarrow^{\mathcal{E}_2} \left(P * (Q \twoheadrightarrow {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_1 \cup \mathcal{E}_f} R)\right).$$

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 $\boxed{P_1}^{\mathcal{N}.\mathrm{lock}}$ and $\boxed{P_2}^{\mathcal{N}.\mathrm{counter}}$, 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.

[17]Note that we have already used namespaces implicitly when specifying, *e.g.*, the counter with contributions module in Section 7. Specifically in the definition of the isCounter predicate (27) on page 74.

13 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*.[18] 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, or refer to the paper [7] on the semantics of Iris.

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.[19]

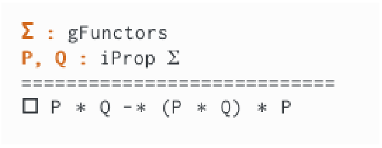
As a first example we prove specification from Example 7.4. This is the simplest non-trivial concurrent example. It uses invariants, but no ghost state. As a second example we prove counter specifications from Section 7.7, and using the precise counter specification we show a proof of the client from Exercise 7.51. 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.

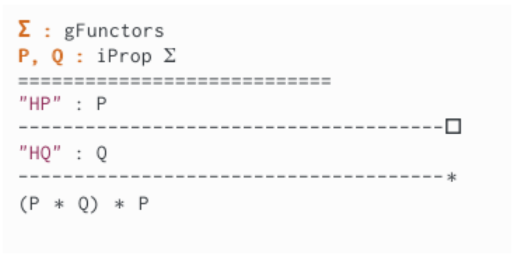

```
Σ : gFunctors
P, Q : iProp Σ
============================
□ P ∗ Q -∗ (P ∗ Q) ∗ P
```

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.

[18] The original paper describing the interactive proof mode is Krebbers *et al.* [10], but the proof mode has evolved significantly since then, and a lot of the tactics described therein are superseded.

[19] gitlab.mpi-sws.org/FP/iris-examples/tree/master/theories/lecture_notes

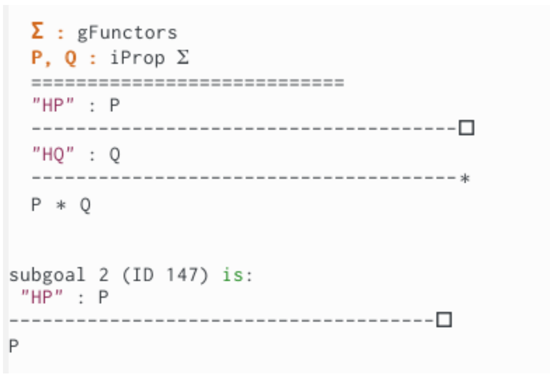


```
Σ : gFunctors
P, Q : iProp Σ
============================
"HP" : P
--------------------------------------□
"HQ" : Q
--------------------------------------∗
(P ∗ Q) ∗ P
```

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 assumptions are named in ordinary Coq proofs, except that for engineering reasons names of Iris assumptions need to be quoted as strings. This is the context of *persistent assumptions*. Every assumption in this context implicitly has an $\square$ 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 $\square P$, and then use the separating conjunction introduction rule. There is a tactic which corresponds to the separating conjunction introduction rule,[20] 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

$$\{P\}\,e\,\{\Phi\} \triangleq \square(P \twoheadrightarrow \mathsf{wp}\,e\,\{\Phi\}).$$

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

$$\{P\}\,e\,\{\Phi\}_{\mathcal{E}} \triangleq \square(\forall \Psi, P \twoheadrightarrow \triangleright(\forall v, \Phi(v) \twoheadrightarrow \Psi(v)) \twoheadrightarrow \mathsf{wp}_{\mathcal{E}}\,e\,\{\Psi\}).$$

[20]There are in fact two, isSplitL, and iSplitR.

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 13.1. Show that for expressions *e which are not values* the two definitions are logically equivalent. $\diamond$

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.

14 Case Study: Types and Abstraction: Logical Relations in Iris

This section is still work-in-progress. It is closely based on [15]; indeed large parts of this section are taken directly from [15], with permission from the authors.

So far we have used Iris for specifying and reasoning about programs written in the *untyped* programming language $\lambda_{\text{ref,conc}}$. In this section we will consider a *typed* programming language, $\mathsf{F}_{\mu,ref,conc}$, and show how we can use Iris to give semantic interpretations of the types of $\mathsf{F}_{\mu,ref,conc}$.

In the words of Reynolds [13], a type system "is a syntactic discipline for enforcing levels of abstraction". One of the fundamental properties of a type system is type soundness, *i.e.*, that "syntactically well-typed programs don't go wrong" [11]. In contrast to the properties we have considered earlier in these notes, where we so far have focused on properties of individual programs, type soundness is a language property: it is a property that depends on the operational semantics and the type system for the whole programming language and it is a property that one proves once and for all for a programming language. There are different approaches to proving type soundness, most notably the syntactic approach based on progress and preservation lemmas (for textbook treatments, see, e.g., [12, 3]), and the semantic approach where one gives a semantic model of the types and proves that any syntactically well-typed program is in the semantic interpretation of its type.

An advantage of the semantic approach is that it gives a semantic account of the invariants enforced by the syntactic type system. This means that it allows one to *combine* syntactically well-typed programs with programs which are semantically, but not necessarily syntactially, well-typed. This is important in practise since most realistic statically typed programming languages include some facility for interacting with programs that are not syntactically well typed (e.g., through foreign function call interfaces, or through an "unsafe" construct []).

An advantage of the syntactic approach using progress and preservation lemmas is that it scales well to advanced type systems including impredicative polymorphism, recursive types, and general reference types. In contrast, an often-cited challenge with the semantic approach is that it is non-trivial to define semantic interpretations of advanced type systems (because the semantic models need to support recursive definitions and impredicative invariants).

In this section we show that *by interpreting types as Iris propositions it is straightforward to give a semantic interpretation of types*, and prove that all syntactically well-typed programs are in the appropriate semantic interpretation. Moreover, we can also use Iris to show semantic well-typedness of programs that are not syntactically well-typed.

The key point is that we can exploit Iris features such as invariants, persistence, and the possibility of defining predicates by guarded recursion to give a simple inductive definition of the interpretation of all the types of $\mathsf{F}_{\mu,ref,conc}$, which include challenging types such as impredicative polymorpic types, recursive types, and general reference types. We focus on type soundness in Subsection 14.2.

Another advantage of the semantic approach is that it can be generalized to give a *relational interpretation* of types, which is useful for proving contextual refinement and data abstraction results. It is well-known that relational models for showing contextual refinement are also non-trivial to construct. Moreover, it is particularly challenging to construct relational models which are sufficiently powerful to allow one to prove relatedness of programs that use state and concurrency in very different ways. Using Iris, however, we can addresss both of these challenges. Indeed, in [15] a relational interpretation of the types of $\mathsf{F}_{\mu,ref,conc}$ is also defined, by interpreting types as Iris relations. Moreover, it is shown that the relational interpretation is expressive enough to allow one to prove challenging examples of program refinements. For example, it is proved that a fine-grained concurrent stack module is a contextual refinement of a coarse-grained stack module. In light of the fact that we so far only used Iris for reasoning about a single program at a time, it is perhaps somewhat surprising that we can also use Iris to relate two different programs. Thus, in Subsection 14.3 we sketch the key idea of how this can be done. However, we do not include a full description of the relational model, but instead refer the reader to [15].

14.1 The language $\mathsf{F}_{\mu,ref,conc}$

$\mathsf{F}_{\mu,ref,conc}$ is mostly as $\lambda_{\text{ref,conc}}$, the main exception being that it is a typed languages. We have all the same values and expressions as before with a few additions:

- fold and unfold respectively fold and unfold expressions of recursive types.
- Λ is a type-level lambda abstraction.

Syntax

$$
\begin{array}{llll}
\text{Val} & v & ::= & \cdots \mid \mathsf{fold}\, v \\
\mathit{Exp} & e & ::= & \cdots \mid \mathsf{fold}\, e \mid \mathsf{unfold}\, e \mid \Lambda\, e \mid e\, \_ \\
\mathit{ECtx} & E & ::= & \cdots \mid \mathsf{fold}\, E \mid \mathsf{unfold}\, E \mid E\, \_ \\
\mathit{Types} & \tau & ::= & X \mid 1 \mid \mathbb{N} \mid \mathbb{B} \mid \tau \to \tau \mid \forall X.\, \tau \mid \tau \times \tau \mid \tau + \tau \mid \mu X.\tau \mid \mathit{ref}\, \tau
\end{array}
$$

We have the same reductions as $\lambda_{\text{ref,conc}}$with the addition of the following two pure reductions:

$$(\Lambda\, e)\, \_ \overset{\text{pure}}{\leadsto} e$$
$$\mathsf{unfold}\, \mathsf{fold}\, v \overset{\text{pure}}{\leadsto} v$$

As the weakest precondition is defined from the operational semantics, we get the same wp-rules as in Figure 10 with the addition of the following two rules, corresponding to our two new reductions:

$$\frac{\textsc{wp-TLam}}{\triangleright \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, (\Lambda\, e)\, \_ \{\Phi\}} \qquad \frac{\textsc{wp-fold}}{\triangleright \mathsf{wp}_{\mathcal{E}}\, v\, \{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\, \mathsf{unfold}\, (\mathsf{fold}\, v)\, \{\Phi\}}$$

To make $\mathsf{F}_{\mu,ref,conc}$ a typed language, we naturally need some typing rules. These are fairly standard (see [12, 3]) and given in Figure 14.

Our goal is to prove type safety, hence we need a notion of safety. Here we will use the following definition:

T-VAR
$$\frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau}$$

T-UNIT
$$\Xi \mid \Gamma \vdash () : 1$$

T-NAT
$$\Xi \mid \Gamma \vdash n : \mathbb{N}$$

T-BOOL
$$\frac{v \in \{\mathsf{true}, \mathsf{false}\}}{\Xi \mid \Gamma \vdash v : \mathbb{B}}$$

T-REC
$$\frac{\Xi \mid \Gamma, x : \tau, f : \tau \to \tau' \vdash e : \tau \to \tau'}{\Xi \mid \Gamma \vdash \mathsf{rec}\, f(x) = e : \tau \to \tau'}$$

T-APP
$$\frac{\Xi \mid \Gamma \vdash e_1 : \tau \to \tau' \qquad \Xi \mid \Gamma \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash e_1\ e_2 : \tau'}$$

T-TLAM
$$\frac{\Xi, X \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall X.\, \tau}$$

T-TAPP
$$\frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]}$$

T-IF
$$\frac{\Xi \mid \Gamma \vdash e : \mathbb{B} \qquad \Xi \mid \Gamma \vdash e_i : \tau \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau}$$

T-PAIR
$$\frac{\Xi \mid \Gamma \vdash e_1 : \tau_1 \qquad \Xi \mid \Gamma \vdash e_2 : \tau_2}{\Xi \mid \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

T-PROJ
$$\frac{\Xi \mid \Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \pi_i\, e : \tau_i}$$

T-INJ
$$\frac{\Xi \mid \Gamma \vdash e : \tau_i \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \mathsf{inj}_i\, e : \tau_1 + \tau_2}$$

T-MATCH
$$\frac{\Xi \mid \Gamma \vdash e : \tau_1 + \tau_2 \qquad \Xi \mid \Gamma, x : \tau_i \vdash e_i : \tau_3 \qquad i \in \{1,2\}}{\Xi \mid \Gamma \vdash \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_i\, x \Rightarrow e_i\, \mathsf{end} : \tau_3}$$

T-FOLD
$$\frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau/X]}{\Xi \mid \Gamma \vdash \mathsf{fold}\, e : \mu X.\, \tau}$$

T-UNFOLD
$$\frac{\Xi \mid \Gamma \vdash e : \mu X.\, \tau}{\Xi \mid \Gamma \vdash \mathsf{unfold}\, e : \tau[\mu X.\, \tau/X]}$$

T-ALLOC
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{ref}(e) : \mathsf{ref}(\tau)}$$

T-LOAD
$$\frac{\Xi \mid \Gamma \vdash e : \mathsf{ref}(\tau)}{\Xi \mid \Gamma \vdash\ !e : \tau}$$

T-STORE
$$\frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash e_1 \leftarrow e_2 : 1}$$

T-CAS
$$\frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau \qquad \Xi \mid \Gamma \vdash e_3 : \tau \qquad \mathsf{EqType}(\tau)}{\Xi \mid \Gamma \vdash \mathsf{cas}(e_1, e_2, e_3) : \mathbb{B}}$$

EQTYP-UNIT
$$\mathsf{EqType}(1)$$

EQTYP-NAT
$$\mathsf{EqType}(\mathbb{N})$$

EQTYP-BOOL
$$\mathsf{EqType}(\mathbb{B})$$

EQTYP-REF
$$\mathsf{EqType}(\mathsf{ref}(\tau))$$

T-FORK
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{fork}\, \{e\} : 1}$$

Figure 14: The typing rules of $\mathsf{F}_{\mu,ref,conc}$.

Safety We say a program e is safe, written $\mathit{Safe}(e)$, if it does not get stuck. In more detail, e is safe if, for all expressions e' that e or one of the child thread of e has evaluated to, e' is either a value or it can be evaluated further by making a head step, or by forking a thread. This is formally defined as follows:

$$\begin{aligned}\mathit{Safe}(e) \triangleq \forall e', \overrightarrow{e_1}, \overrightarrow{e_2}, \sigma.\ (\emptyset, e) \to_{\mathsf{tp}}^{*} (\sigma, \overrightarrow{e_1}; e'; \overrightarrow{e_2}) \Rightarrow \\ e' \in \mathsf{Val} \vee (\exists e'', \sigma'.\ (\sigma, \overrightarrow{e_1}; e'; \overrightarrow{e_2}) \to_{\mathsf{tp}} (\sigma', \overrightarrow{e_1}; e''; \overrightarrow{e_2})) \vee \\ (\exists e'', \sigma', e_3.\ (\sigma, \overrightarrow{e_1}; e'; \overrightarrow{e_2}) \to_{\mathsf{tp}} (\sigma', \overrightarrow{e_1}; e''; \overrightarrow{e_2}; e_3))\end{aligned}$$

14.2 Unary Logical relation

In this section we define a unary logical relations model for $\mathsf{F}_{\mu,ref,conc}$ and use it to prove the type soundness theorem. That is, for each type we define a logical relation. We show that each well-typed program is in the relation for its type. Furthermore, we show that programs in the logical relation for a type have well-defined behavior, *i.e.*, they do not get stuck. The type soundness theorem is a direct consequence of these two facts.

We define the logical relations in three stages. We first define a relation for *closed* values of a type by induction on types. We then use the value relations to define relations on *closed* expressions. Intuitively, a closed expression is in the relation for a type τ if it is a computation that results in a value that is in the value relation for the type τ. Finally, we define *the logical relation* for (open) expressions based on the expression relations and the value relations above.

$\mathsf{F}_{\mu,ref,conc}$ features polymorphism. Hence, types can have free type variables. Thus, we index the relations on closed values and expressions with a map, Δ, which assigns a semantic type (a value relation) to each free type variable. That is, for each type τ we define the value relation $[\![\Xi \vdash \tau]\!]_\Delta : \mathsf{Val} \to \mathsf{Prop}$ where $\Delta : \Xi \to \mathsf{Val} \to \mathsf{Prop}$. The full definition of the value relations for types is given in Figure 15. We will discuss them in detail below. Before that we discuss how value relations are extended to expression relation on closed and subsequently open expressions.

Intuitively, a closed expression is in the expression relation for a type τ if it computes a result that is in the value relation for the type τ. We define the expression relation, $[\![\Xi \vdash \tau]\!]_\Delta^{\mathcal{E}} : \mathit{Exp} \to \mathsf{Prop}$, using weakest preconditions, as follows:

$$[\![\Xi \vdash \tau]\!]_\Delta^{\mathcal{E}}(e) \triangleq \mathsf{wp}\, e\, \{[\![\Xi \vdash \tau]\!]_\Delta\}$$

In order to formally define the logical relation for open expressions we first define a relation, $[\![\Xi \vdash \cdot]\!]_\Delta^{\mathcal{G}}$, for typing contexts. Intuitively, a list a values, $\vec{v}$, is in the relation for a typing context, Γ, if each value in $\vec{v}$ is in the value relation for the type corresponding to it in Γ. The formal definition of the typing-context relation is given below. Here, ε is the empty list of values.

$$\begin{aligned}[\![\Xi \vdash \cdot]\!]_\Delta^{\mathcal{G}}(\varepsilon) &\triangleq \top \\ [\![\Xi \vdash \Gamma, x : \tau]\!]_\Delta^{\mathcal{G}}(\vec{v}, w) &\triangleq [\![\Xi \vdash \Gamma]\!]_\Delta^{\mathcal{G}}(\vec{v}) * [\![\Xi \vdash \tau]\!]_\Delta(w)\end{aligned}$$

Let Ξ, Γ, e and τ be such that all free variables of e are in the domain of Γ and all free type variables that appear in Γ or τ are in Ξ. Then, we write $\Xi \mid \Gamma \vDash e : \tau$ to express that the expression e is in the logical relation for type τ under the typing contexts Γ and Ξ. This relation is defined as follows:

$$\Xi \mid \Gamma \vDash e : \tau \triangleq \forall \Delta, \vec{v}.\ [\![\Xi \vdash \Gamma]\!]_\Delta^{\mathcal{G}}(\vec{v}) \vdash [\![\Xi \vdash \tau]\!]_\Delta^{\mathcal{E}}(e[\vec{v}/\vec{x}])$$

The value relation for types are given in Figure 15. One important aspect of the type system of $\mathsf{F}_{\mu,ref,conc}$ is that it is intuitionistic, *i.e.*, values, e.g., function arguments, can be used multiple

$$
\begin{aligned}
[\![\Xi \vdash X]\!]_\Delta &\triangleq \Delta(X)\\
[\![\Xi \vdash 1]\!]_\Delta(v) &\triangleq v = ()\\
[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v) &\triangleq \exists n \in \mathbb{N}.\, v = n\\
[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v) &\triangleq v \in \{\mathsf{true}, \mathsf{false}\}\\
[\![\Xi \vdash \tau_1 \times \tau_2]\!]_\Delta(v) &\triangleq \exists v_1, v_2.\, v = (v_1, v_2) * [\![\Xi \vdash \tau_1]\!]_\Delta(v_1) * [\![\Xi \vdash \tau_2]\!]_\Delta(v_2)\\
[\![\Xi \vdash \tau_1 + \tau_2]\!]_\Delta(v) &\triangleq \bigvee_{i \in \{1,2\}} \exists w.\, v = \mathsf{inj}_i\, w * [\![\Xi \vdash \tau_i]\!]_\Delta(w)\\
[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v) &\triangleq \Box\Big(\forall w.\, [\![\Xi \vdash \tau]\!]_\Delta(w) \mathrel{-\!\!*} [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta(v\ w)\Big)\\
[\![\Xi \vdash \mu X.\, \tau]\!]_\Delta(v) &\triangleq \mu\Psi.\, \exists w.\, v = \mathsf{fold}\, w \wedge \triangleright [\![\Xi \vdash \tau]\!]_{\Delta, X \mapsto \Psi}(w)\\
[\![\Xi \vdash \forall X.\, \tau]\!]_\Delta(v) &\triangleq \Box\Big(\forall \Psi.\, \mathsf{persistent}(\Psi) \Rightarrow [\![\Xi, X \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X \mapsto \Psi}(v\ \_)\Big)\\
[\![\Xi \vdash \mathsf{ref}(\tau)]\!]_\Delta(v) &\triangleq \exists \ell.\, v = \ell \wedge \boxed{\exists w.\, \ell \mapsto w * [\![\Xi \vdash \tau]\!]_\Delta(w)}^{\mathcal{N}.\ell}
\end{aligned}
$$

Figure 15: The unary value relation for types of $\mathsf{F}_{\mu,ref,conc}$.

times. Thus, it is crucial that the value relation of all types are *persistent* and hence duplicable. The persistence modality and the side-condition $\mathsf{persistent}(\Psi)$ in Figure 15 are added to ensure the persistence of value relations.

The value relation for type variables is given by Δ. A value is in the relation for the unit type if it is (). A values is in the relation for the type of natural numbers if it is simply a natural number; similarly for booleans. A value is in the relation for the product type if it is a pair of values each in their respective types. A value of the sum type $\tau + \tau'$, on the other hand, is either a value in the relation for τ or one in the relation for τ'. The value relation for recursive types is defined using Iris's guarded recursive predicates. A value is in the relation for a recursive type if it is of the form $\mathsf{fold}\, w$ such that the value w is, *one step of the computation later*, in the relation for the recursive type. Notice, however, that unfolding a folded value takes a step of computation. A memory location is in the relation for a reference type, $\mathsf{ref}(\tau)$, if it *invariantly* stores a value that is in the value relation for τ.

A value v in the relation for the function type $\tau \to \tau'$ if whenever v is applied to a value w, in the relation for τ, the resulting expression, $v\ w$, is in the *expression* relation for τ'. A value is in the relation for the type $\forall X.\, \tau$ if, when instantiated, the resulting *expression* is in the expression relation for τ where the interpretation for X is taken to be any *persistent* predicate.

Lemma 14.1. *Let τ be a type such that all its free type variables appear in Ξ. Furthermore, let X be a type variable such that $X \notin \Xi$ and let Δ be an interpretation for type variables in Ξ. It follows that*

$$[\![\Xi \vdash \tau]\!]_\Delta(v) \dashv\vdash [\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto \Psi}(v)$$

for any predicate Ψ and value v.

Proof. By induction on the structure of τ. □

Lemma 14.2. *Let Γ be a typing context such that all free type variables of Γ appear in Ξ. Furthermore, let X be a type variable such that $X \notin \Xi$ and let Δ be an interpretation for type variables in Ξ. It follows that*

$$[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v}) \dashv\vdash [\![\Xi, X \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta, X \mapsto \Psi}(\vec{v})$$

for any predicate Ψ and sequence of values $\vec{v}$.

Proof. By induction on the length of Γ using Lemma 14.1. $\square$

Theorem 14.3 (Fundamental theorem of unary logical relations)**.** *All well-typed terms are in the logical relation.*

$$\textit{If } \Xi \mid \Gamma \vdash e : \tau \textit{ then } \Xi \mid \Gamma \vDash e : \tau$$

Proof. By induction on the typing derivation. All cases follow from the inference rules of weakest preconditions presented in Section 12.1. Here, we present a few cases of this proof.

- Case T-ALLOC: For this case, given $\Delta : \Xi \to \mathsf{Val} \to \mathsf{Prop}$ and a list of values $\vec{v}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v})$, we need to show assuming

$$\mathsf{wp}\, e[\vec{v}/\vec{x}]\,\{[\![\Xi \vdash \tau]\!]_{\Delta}\} \tag{50}$$

that the following holds:

$$\mathsf{wp}\, \mathsf{ref}(e[\vec{v}/\vec{x}])\,\{x.\exists \ell.\ x = \ell \land \boxed{\exists w.\ \ell \mapsto w * [\![\Xi \vdash \tau]\!]_{\Delta}(w)}^{\mathcal{N}.\ell}\}$$

We use the rule WP-BIND together with the assumption (50) above. Consequently, we need to show that given some arbitrary value v such that

$$[\![\Xi \vdash \tau]\!]_{\Delta}(v) \tag{51}$$

we have

$$\mathsf{wp}\, \mathsf{ref}(v)\,\{x.\exists \ell.\ x = \ell \land \boxed{\exists w.\ \ell \mapsto w * [\![\Xi \vdash \tau]\!]_{\Delta}(w)}^{\mathcal{N}.\ell}\}$$

We proceed by applying the rule WP-ALLOC which requires us to show:

$$\triangleright \forall \ell'.\ \ell' \mapsto v \mathrel{-\!\!*} \mathsf{wp}\, \ell'\,\{x.\exists \ell.\ x = \ell \land \boxed{\exists w.\ \ell \mapsto w * [\![\Xi \vdash \tau]\!]_{\Delta}(w)}^{\mathcal{N}.\ell}\}$$

This follows easily from the rules WP-VAL and INV-ALLOC together with assumption (51) above.

- Case T-REC: For this case, given $\Delta : \Xi \to \mathsf{Val} \to \mathsf{Prop}$ and a list of values $\vec{v}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta}(\vec{v})$, we need to show assuming

$$\forall v, u.\ [\![\Xi \vdash \tau]\!]_{\Delta}(v) * [\![\Xi \vdash \tau \to \tau']\!]_{\Delta}(u) \mathrel{-\!\!*} \mathsf{wp}\, (e[\vec{v}/\vec{x}])[v, u/x, f]\,\{[\![\Xi \vdash \tau']\!]_{\Delta}\} \tag{52}$$

that the following holds:

$$\mathsf{wp}\, \mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}]\,\{x.\ \square\Big(\forall w.\ [\![\Xi \vdash \tau]\!]_{\Delta}(w) \mathrel{-\!\!*} [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_{\Delta}(x\ w)\Big)\}$$

By the rule WP-VAL it suffices to show:[21]

$$\forall w.\ [\![\Xi \vdash \tau]\!]_{\Delta}(w) \mathrel{-\!\!*} [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_{\Delta}((\mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}])\ w)$$

Note the operational semantics pertaining to calling a recursive functions. The expression

$$(\mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}])\ w$$

[21]We can introduce the persistence modality because all assumptions are persistent.

reduces to the expression

$$(e[\vec{v}/\vec{x}])[w, \mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}]/x, f]$$

in a single step of computation. Hence, if we know that $[\![\Xi \vdash \tau \to \tau']\!]_\Delta(\mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}])$ holds we can use the assumption (52) above to finish the proof. However, this is exactly what we have to show but crucially we only need this *after one step of computation*, intuitively because a recursive call can only occur inside the body, after the current call. Hence, to finish the proof we use the Löb rule. Consequently we get to assume the following Löb induction hypothesis (IH).

$$\triangleright \forall w.\ [\![\Xi \vdash \tau]\!]_\Delta(w) \twoheadrightarrow [\![\Xi \vdash \tau']\!]^{\mathcal{E}}_\Delta((\mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}])\ w) \tag{IH}$$

We finish the proof using the rule WP-REC which requires us to show the following for some arbitrary value w for which we have $[\![\Xi \vdash \tau]\!]_\Delta(w)$:

$$\triangleright \mathsf{wp}\,(e[\vec{v}/\vec{x}])[w, \mathsf{rec}\, f(x) = e[\vec{v}/\vec{x}]/x, f]\{[\![\Xi \vdash \tau']\!]_\Delta\}$$

This follows easily from our assumptions and the Löb induction hypothesis, (IH), above.

– Case T-TLAM: For this case, given $\Delta : \Xi \to \mathsf{Val} \to \mathsf{Prop}$ and a list of values $\vec{v}$ such that $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v})$, we need to show assuming

$$\forall \Psi.\ [\![\Xi, X \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta, X \mapsto \Psi}(\vec{v}) \vdash \mathsf{wp}\, e[\vec{v}/\vec{x}]\{[\![\Xi, X \vdash \tau']\!]_{\Delta, X \mapsto \Psi}\} \tag{53}$$

that the following holds:

$$\mathsf{wp}\, \Lambda\, e[\vec{v}/\vec{x}]\{x.\ \Box\Big(\forall \Psi.\ \mathsf{persistent}(\Psi) \Rightarrow [\![\Xi, X \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X \mapsto \Psi}(v\ \_)\Big)\}$$

Since $\Lambda\, e[\vec{v}/\vec{x}]$ is a value, we use the rule WP-VAL. Hence, it suffices to show the following for some arbitrary but fixed Ψ such that $\mathsf{persistent}(\Psi)$:

$$[\![\Xi, X \vdash \tau]\!]^{\mathcal{E}}_{\Delta, X \mapsto \Psi}((\Lambda\, e[\vec{v}/\vec{x}])\ \_)$$

Note that here we can introduce the persistence modality as none of our assumptions assert any ownership. Unfolding the expression relation in the above formula reveals that we need to show:

$$\mathsf{wp}\,(\Lambda\, e[\vec{v}/\vec{x}])\ \_\{[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto \Psi}\}$$

To prove this, we proceed by applying the rule WP-TLAM and as a result need to show:

$$\triangleright \mathsf{wp}\, e[\vec{v}/\vec{x}]\{[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto \Psi}\}$$

Finally, we can finish the proof by appealing to assumption (53). We only need to show $[\![\Xi, X \vdash \Gamma]\!]^{\mathcal{G}}_{\Delta, X \mapsto \Psi}(\vec{v})$ while we have $[\![\Xi \vdash \Gamma]\!]^{\mathcal{G}}_\Delta(\vec{v})$. However, this follows from Lemma 14.2.

□

Lemma 14.4 (Adequacy of unary logical relations)**.** *Let e be an expression such that $[\![\Xi \vdash \tau]\!]^{\mathcal{E}}_\Delta(e)$. Then, e is safe, $Safe(e)$.*

Proof. This lemma is a direct consequence of the adequacy theorem, see [7]. □

Theorem 14.5 (Soundness of unary logical relations)**.** *All closed well-typed programs of* $\mathsf{F}_{\mu,ref,conc}$ *are safe:*

$$\textit{If} \;\; \cdot \mid \cdot \vdash e : \tau \;\; \textit{then} \;\; \textit{Safe}(e)$$

Proof. By the fundamental theorem of unary logical relation, Theorem 14.3, we know that

$$\cdot \mid \cdot \vDash e : \tau$$

Expanding the definition of unary logical relations we get

$$\forall \Delta, \vec{v}.\ [\![\cdot \vdash \cdot]\!]^{\mathcal{G}}_{\Delta}(\vec{v}) \vdash [\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e[\vec{v}/\vec{x}])$$

We take $\Delta = \emptyset$ and $\vec{v} = \varepsilon$ which gives us

$$[\![\cdot \vdash \cdot]\!]^{\mathcal{G}}_{\emptyset}(\varepsilon) \vdash [\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e[\varepsilon/\varepsilon])$$

which immediately simplifies to $[\![\cdot \vdash \tau]\!]^{\mathcal{E}}_{\Delta}(e)$. By Theorem 14.4, we get *Safe*(e), as required. □

The logical relations model presented in this section is modular. For instance, a well-typed function of type $\tau \to \tau'$ (which by the fundamental theorem falls in the logical relation) can be applied to any expression that is the logical relations for τ and the result is in the logical relation for τ'. On the other hand, our logical relations model is defined in terms of untyped expressions. This means that our logical relations model can be used to prove safety of programs that mix well-typed code with untyped code as long as we show that the untyped code is *semantically well-typed*, *i.e.*, the untyped code is in the logical relations for the appropriate type. As an example of a program that is *not* well-typed but is nonetheless semantically well-typed consider the following:

$$\begin{aligned}\Lambda\,\Lambda\,(\lambda f.\,\lambda x.\,\lambda g.\,\lambda y.\,&\mathsf{let}\, l = \mathsf{ref}(\mathsf{true})\,\mathsf{in}\,\mathsf{fork}\,\{l \leftarrow \mathsf{false}\};\\ &\mathsf{if}\,!\,l\,\mathsf{then}\ \text{waitfor}\ l; l \leftarrow x; \mathsf{inj}_1\,(f\ l)\,\mathsf{else}\,l \leftarrow y; \mathsf{inj}_2\,(g\ l))\end{aligned}$$

where

$$\text{waitfor} \triangleq \mathsf{rec}\, f(x) = \mathsf{if}\,!\,x\,\mathsf{then}\, f\ x\,\mathsf{else}\,()$$

This program does not syntactically have the type

$$\forall X. \forall Y. (\mathsf{ref}(X) \to X) \to X \to (\mathsf{ref}(Y) \to Y) \to Y \to X + Y$$

but it *semantically* does. This program allocates a boolean reference and uses it to non-deterministically call f or g. In each case it changes the reference that it has already allocated with the given value of the appropriate type before passing it to the chosen function. In case the decision is made to call the first function, *i.e.*, the other thread has not succeeded in the race, it waits for the other thread to finish. This is to ensure that the other thread writing to the reference l is not going to destroy the contents of l at some later point. Despite not being syntactically well-typed one can show that the program above is semantically of the type given. Hence, this program can be *safely* linked against any other (syntactically or semantically) well-typed program with compatible type.

14.3 Binary logical relation for contextual refinement

In [15] the authors also define a binary logical relations model for $\mathsf{F}_{\mu,ref,conc}$ and prove that logical relatedness implies contextual refinement.

Many of the ideas from the unary case carry over to the binary case, *e.g.*, the binary value interpretation is a straightforward modification of the unary value interpretation. The expression interpretation, however, is quiet different: in the unary case, it sufficed to use a simple weakest precondition definition, but now, in the binary case, the challenge is to find a way to relate two (different) expressions. In this section, we sketch the key ideas of how one can define a relational model. We do not include the full definition of the binary logical relation, but instead refer the reader to [15].

In the relational model, the goal is to define a logical relation, such that if e is logically related to e' then e contextually refines e'. We often refer to the expression "on the left", e, as the implementation side expression and the expression "on the right", e', as the specification side expression. To define the relational models, we need a way to refer to (the heap and different threads of) the program on the specification side "that is about to be executed". The intuitive definition of two expressions being related is as follows:

An expressions e (the implementation side) is related to an expression e′ (the specification side) if we have:

$$\begin{aligned}&\forall j, E.\ \text{“}\textit{thread } j \textit{ is about to execute } E[e']\text{”} \twoheadrightarrow \\ &\qquad \mathit{wp}\, e \{\exists v'.\ \text{“}\textit{thread } j \textit{ is about to execute } E[v']\text{”}\}\end{aligned}$$

This relation between e and e' reads as follows: if thread j is about to execute e' under some evaluation context E on the specification side and e reduces to a value, then there is a value v' such that the specification side is about to execute $E[v']$. In other words, whenever e reduces to a value we know that e' has also been reduced to v'. The reason for explicit quantification over the thread j under which the specification side is being executed is to enable thread-local reasoning. We quantify over the evaluation context E under which the expression is about to be executed to enable modular reasoning with respect to evaluation contexts.

The goal is now to be able to formalize that "thread j is about to execute $E[e']$". For this, we first define the monoids:

$$\begin{aligned}\textsc{Heap} &\triangleq \textsc{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} (\mathrm{Ex}(\mathrm{Val})))\\ \textsc{Tpool} &\triangleq \textsc{Auth}(\mathbb{N} \xrightarrow{\text{fin}} (\mathrm{Ex}(\mathit{Exp})))\end{aligned}$$

Letting γ'_h, γ'_{tp} be instances of Heap[22] respectively Tpool we now define the following propositions:

$$\begin{aligned}\mathrm{SpecConf}(\sigma, \vec{e}) &\triangleq \boxed{\bullet\, \mathrm{res}(\sigma)}^{\gamma'_h} * \boxed{\bullet\, \mathrm{fpfnOf}(\vec{e})}^{\gamma'_{tp}}\\ \ell \mapsto_s v &\triangleq \boxed{\circ\, [\ell \mapsto v]}^{\gamma'_h}\\ j \Mapsto_s e &\triangleq \boxed{\circ\, [j \mapsto e]}^{\gamma'_{tp}}\end{aligned}$$

[22]Notice that is the same resource algebra, that is used to define the standard $\mapsto$ predicate. Hence we now have two instances of it, one γ_h for tracking the heap on the implementation side and one γ'_h for tracking the heap on the specification side.

where

$$\mathrm{fpfnOf}(e_1, \ldots, e_n) \triangleq \{(i, e_i) | 1 \leq i \leq n\}$$

Here, ρ is a configuration, *i.e.*, a pair of a heap and a thread pool, $(\sigma, \vec{e})$. The intuition is that $\mathrm{SpecConf}(\rho)$ is the (unique) configuration that the specification side is in. The propositions $j \Rrightarrow e$ and $\ell \mapsto_s v$ simply specify the exclusive ownership of the execution of a thread or a memory location on the specification side. Naturally, this requires the propositions to be exclusive.

Furthermore we use the following invariant to express that the specification side is in a configuration reachable from some starting configuration:

$$\mathrm{SpecCtx}(\rho) \triangleq \boxed{\exists \rho'. \, \mathrm{SpecConf}(\rho') \land \rho \rightarrow^*_{\mathrm{tp}} \rho'}^{\mathcal{N}.sc}$$

With these tools, it is possible to define the expression relation as:

$$\begin{aligned}\llbracket \Xi \vdash \tau \rrbracket^{\mathcal{E}}_{\Delta}(e, e') \triangleq \forall \rho, j, E. \, & \mathrm{SpecCtx}(\rho) * j \Rrightarrow E[e'] \mathrel{-\!\!*} \\ & \mathsf{wp}\, e \,\{v. \, \exists v'. \, j \Rrightarrow E[v'] * \llbracket \Xi \vdash \tau \rrbracket_{\Delta}(v, v')\}\end{aligned}$$

The expression relation above states that e and e' are related if the following holds: for any thread j which is about to execute e' under some evaluation context E, it is safe to evaluate e and whenever e reduces to a value v, we know that e' has also been evaluated to some value v' in thread j under the evaluation context E. Furthermore, we know that v and v' will be related as values of the type relating e and e'. Thus, essentially, two expressions e and e' are related at type τ if, whenever e reduces to a value, so does e' (no matter under which circumstances it is being evaluated), and the resulting values will be related at type τ.

One can now define the binary logical relations for $\mathsf{F}_{\mu,ref,conc}$, written $\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau$, as follows:

$$\Xi \mid \Gamma \vDash e \leq_{log} e' : \tau \triangleq \forall \vec{v}, \vec{v'}, \Delta. \, \llbracket \Xi \vdash \Gamma \rrbracket^{\mathcal{G}}_{\Delta}(\vec{v}, \vec{v'}) \vdash \llbracket \Xi \vdash \tau \rrbracket^{\mathcal{E}}_{\Delta}(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$

where $\vec{x}$ is the domain of Γ.

See [15] for the definition of the value relation and also for applications of the logical relation for proving challenging contextual refinements.

References

[1] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231, 2014. 112

[2] Thomas Dinsdale-Young, Pedro da Rocha Pinto, and Philippa Gardner. A perspective on specifying and verifying concurrent modules. *Journal of Logical and Algebraic Methods in Programming*, 98:1 – 25, 2018. 112

[3] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016. 138, 139

[4] 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

[5] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011. 112

[6] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016. 1

[7] 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. *Journal of Functional Programming*, 2018. 1, 50, 136, 144

[8] 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, 119

[9] 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

[10] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 205–217, New York, NY, USA, 2017. ACM. 136

[11] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978. 138

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. 138, 139

[13] John C. Reynolds. Types, abstraction, and parametric polymorphism. *Information Processing*, 1983. 138

[14] K. Svendsen, L. Birkedal, and M. Parkinson. Modular reasoning about separation of concurrent data structures. In *Proceedings of ESOP*, 2013. 112, 119

[15] A. Timany and L. Birkedal. Type soundness and contextual refinement via logical relations in higher-order concurrent separation logic. 2018. Manuscript. 138, 139, 146, 147

[16] 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