Towards Modular Reasoning for Stateful and Concurrent Programs

Morten Krogh-Jespersen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Towards Modular Reasoning for Stateful and Concurrent Programs

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Morten Krogh-Jespersen
September 4, 2018

Abstract

Software is an integral part of our everyday lives and we rely on man-written programs to solve a wide range of problems. Ensuring that programs solve well-defined problems satisfactory can be accomplished by the art of *software verification*, *i.e.* formal reasoning about the program in some mathematically founded model. However, formal reasoning about real-world programs is well known to be difficult because of the advanced programming language features used when writing them.

The main objective of this dissertation is to develop *state-of-the-art* models that allow for verification of programs using advanced language features such as (1) higher-order functions, (2) higher-order store (general references) and (3) concurrency. Conceptually, languages with store and concurrency are extremely hard to reason about because threads may race when trying to read and update references. Technically, reasoning about such languages formally require sophisticated mathematical models and some notion of ownership and invariants.

In this dissertation we present a relational model for reasoning about a concurrent, higher-order language with general references, where all references are tracked by a type-and-effect system. The model validates data-abstraction by masking effects and parallelizing expressions if effects are suitable disjoint. Masking stateful effects can in pure languages be done monadically as well. In this dissertation we further present a logical relations model that semantically verifies that runST, the run-function of the ST monad, provides proper encapsulation of state for real-world implementations. Finally, we also present Aneris, a logical framework for writing and reasoning about distributed systems. Aneris allow for node-local reasoning and each node can have local state and concurrency. We show that the framework is suitable for verifying distributed systems by verifying a broad range of interesting examples.

Resumé

Software er en fast integreret del af vores hverdagsliv, og vi har næsten blind tillid til, at en lang række af vores problemer løses korrekt af menneskeskabte programmer. En måde at sikre at disse programmer rent faktisk løser de problemer, hvortil de med formål blev skabt, kan gøres ved hjælp af *software verifikation*, i.e. ved at argumentere formelt for korrekthed i en matematisk funderet model. Desværre er det alment kendt, at det er meget vanskeligt at argumentere for korrektheden af virkelige programmer pga. de avancerede konstruktioner, man kan bruge ved udviklingen af disse programmer.

Det primære fokus for denne afhandling er på udviklingen af splinternye modeller, der muliggør verifikation af programmer, der anvender avancerede programmeringssprogskonstruktioner såsom (1) højere-ordensfunktioner, (2) højere-ordenslager og (3) flertrådet programmering. Konceptuelt er det svært at ræsonnere om flertrådede programmer, der også anvender datalageret, da tråde kan konkurrere om at læse og skrive til referencer. Teknisk set kræver formel verifikation af sådanne programmer avancerede matematiske modeller samt begreber om ejerskab og invarianter.

I denne afhandling præsenteres en relationel model, der tillader at ræssonere om et flertrådet, højere-ordensprogrammeringssprog hvor hukommelseslageret også er højere-orden og hvor alle referencer spores gennem et type-og-effekt system. Modellen validerer data-abstraktion ved at skjule effekter samt parallel sammensætning af programmer, såfremt adgange til data-lageret er tilpas opdelt.

Det er også muligt at skjule effekter monadisk i siddeeffektfrie højere-ordenssprog. I denne afhandling præsenteres også en logisk relationel model, der indfanger, at konstruktionen runST, kørselsfunktionen tilhørende ST monaden, på passende vis indkapsler tilstand for realistiske programmer.

Slutteligt præsenterer afhandlingen den logiske platform Aneris, der er specifikt designet til at skrive og ræsonnere om distribuerede systemer. Aneris tillader brugen af lokal tilstand og tråde, og at man verificerer knuder i isolation, såkaldt knude-lokal-ræsonnering. Det vises, gennem verifikationen af en bred vifte af eksempler, at platformen er et stærkt værktøj ved verifikation af distribuerede systemer.

Acknowledgments

First and foremost I would like to thank my supervisor Lars Birkedal for taking a chance on me as his PhD student, his expert guidance throughout the years and the sense of comradery extended by him. I have always felt Lars has done his best to aid me in my career and for that I am truly grateful.

I am thankful for all of my coauthors valuable input and collaboration, in particular I would like to thank Amin Timany for hosting me in my trip to Leuven and for all of his invaluable help and friendship. I would also like to thank the Logic and Semantics group for all the great discussions, social gatherings and for enabling a good work environment. I hope you will have many trips to the coffee-room after my departure.

A special thanks to Aleš Bizjak for always being helpful and having time for my questions. I wish you all the best in the future. I would also like to thank William Hesse from Google Aarhus for hosting me as an intern in 2017.

Finally, my biggest gratitude goes to my wife Marianne for her generous love and support and for delaying my thesis by giving birth to our second child. Thank you Hannah and Emilie for all your smiles and support through difficult moments and monads. Love from a family may be one of the best *categories (therapy) for the working computer scientist*.

Morten Krogh-Jespersen,
Aarhus, September 4, 2018.

Contents

Part I

Overview

Chapter 1

Introduction

Programming languages with advanced features, such as concurrency, higher-order store and network primitives are standard today, *e.g.* OCaml, Rust, Haskell, Swift to name a few. However, formal reasoning about programs utilizing those advanced features is difficult and very much non-standard.

In this dissertation I present work that facilitates *modular* reasoning about programs with local state, concurrency and network primitives in unary or relation models. Such formally verified models allow developers to formally prove correctness of programs and modules and in some cases allow for automatic compile time optimizations.

The reader is assumed to be familiar with separation logic and some knowledge regarding logical relational models would be beneficial. Futhermore, some familiarity with the proof-assistant Coq is required to understand the formal developments accompanying this dissertation.

1.1 Published Papers and Manuscripts

The following papers are included in Part II, Publications and Manuscripts of this dissertation.

[42] *A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic.*
Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal.
Proceedings of the ACM on Programming Languages (POPL), 2017.
Included in Chapter 3.

[75] *A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST.*
Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal.
Proceedings of the ACM on Programming Languages (POPL), 2018. Included in Chapter 4.

Minor layout adjustments and typo corrections have been carried out on the above papers. In addition to the above published papers, this dissertation also contains the following manuscripts:

- *Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems.* Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, and Lars Birkedal. Included in Chapter 5.

- *Verifying a concurrent data-structure from the Dartino Framework.* Morten Krogh-Jespersen, Thomas Dinsdale-Young, and Lars Birkedal. Included in Chapter 6.

The author of this dissertation have, with the exception of [75], contributed significantly to the above research projects – from the technical innovations, to proving and paper writing. For [75], the author aided in the development and discussions of the logical relation and proofs of the fundamental theorem, along with a proportional writing of the paper.

1.2 Outline of the Dissertation

This dissertation is divided into two parts. Part I provides an overview of the technical developments gathered from the papers and manuscripts. Part II consists of the co-authored papers and manuscripts listed in the previous section.

For Part I, Overview the main chapter is Chapter 2 that summarizes challenges and the scientific contributions for each research project. Additionally, the closest related work is also discussed to allow the reader to better position our technical developments.

For Part II, Publications and Manuscripts, each work has its own chapter. Chapter 3 present the work on a relational model for a type-and-effect system. The chapter also gives a general introduction to invariants and resources in Iris, and uses these basic definitions to gradually build a relational model in a higher-order, separation logic, that validates the parallelization theorem. That is done by starting from a unary relation that characterizes type-inhabitance and introducing necessary resources along the way. Finally, the proof outline of the parallelization theorem is shown.

Chapter 4 considers effectful computations in the spirit of the ST monad by defining a call-by-value language with the operations of the ST as primitives. To reason about effectful computations in a purely fashion way, a logical relational is developed that allow for encapsulation of effects. The logical relational is built on top of several novel logical connectives also explained in detail. Finally, proof outlines of several relational properties and a state-independence theorem is shown.

Chapter 5 presents Aneris, a verification framework for distributed systems built to reason about real-world programs that use sockets. The chapter describes the novelties of Aneris at a high-level and introduce the term *node-local* reasoning. We then formally define the language and logic accompanying the framework. Finally, the logic is used to show two interesting examples: a load-balancer and replicated-logging, which is a client built on top of the consensus protocol two-phase commit.

Finally, Chapter 6, presents a case study for verifying a concurrent process queue from the Dartino framework, which is a virtual machine for running Dart code on IoT-devices. The case study describes the difficulty of formally specifying a process queue before introducing the logical machinery needed. The chapter also discusses the trade-offs by translating the code from C++ to Iris. Finally, the case-study describe how specifications on the process queue's operations can be strengthened by introducing logical atomic triples.

Chapter 2

Contributions of this Dissertation

In this chapter, we present an overview of the main challenges and contributions of the listed work in §1.1. For a full account see Part II, Publications and Manuscripts.

2.1 Relational Model of Types-and-Effects

Programming and reasoning about higher-order, concurrent programs with effects is known to be challenging. As a consequence, different kinds of refined type-systems has been proposed to simplify reasoning about effectful programs. Examples of such type-systems include: alias types [71], capability type systems [60], linear type systems [23, 40, 53] Hoare type theory [54], permissions-based type systems [61], type-and-effect systems [11, 12, 26, 51], etc. There has also been larger-scale implementation efforts on higher-order programming languages, e.g., the Mezzo programming language [61] and the Rust programming language [67], which employ refined type systems to control the use of state in the presence of concurrency.

In Krogh-Jespersen et al. [42], included in Chapter 3, the main technical result is a logical relations model, $\mathrm{LR}_{\mathrm{PAR}}$, that models an expressive region-based type-and-effect system for a higher-order concurrent programming language with general references, $\lambda_{ref,conc}$, that allow for:

- Verifying effect-based transformations and optimizations, including the tricky parallelization theorem.
- Verifying implementations of abstract data types with local state.
- Verifying statically ill-typed terms that satisfy semantic invariants.

It is the first model that allow for the above verification properties for such an expressive language and it is presented in all its gory details in Figure 39. In

the remainder of this section, we discuss the challenges, key ideas and related work for [42].

The Language and the Type-and-Effect System

$\lambda_{ref,conc}$ is standard call-by-value language with general references (higher-order store), dynamic allocation, parallel composition ($\|$) and **CAS** (compare-and-set). An example of a stack module written in $\lambda_{ref,conc}$ is shown in Figure 21.

$$
\begin{aligned}
stack() &= \textsf{let } h = \textsf{new } (\textsf{new } \textsf{inj}_1\ ())\ \textsf{in } (push_2(h), pop_2(h)) \\
push(h) &= \textsf{rec } loop(n).\textsf{let } v = !h \textsf{ in} \\
&\qquad \textsf{if CAS}(h, v, \textsf{new } \textsf{inj}_2\ (n, v))\ \textsf{then } ()\ \textsf{else } loop(n) \\
pop(h) &= \textsf{rec } loop(\_).\textsf{let } v = !h \textsf{ in} \\
&\qquad \textsf{case}(!v, \textsf{inj}_1\ () \Rightarrow \textsf{inj}_1\ () \\
&\qquad\qquad \textsf{inj}_2\ (n, v'') \Rightarrow \textsf{if CAS}(h, v, v'')\ \textsf{then inj}_2\ n\ \textsf{else } loop())
\end{aligned}
$$

Figure 21: An implementation of a stack-module with local references in $\lambda_{ref,conc}$.

The stack module uses **CAS** in its push and pop method, to ensure it functions correctly in the presence of other interfering threads.

We use a type-and-effect system similar to the type-and-effect system in [14], to assign types to expressions in $\lambda_{ref,conc}$. The type-and-effect system is inspired by Lucassen and Gifford's seminal work [26, 51], with the addition that public and private regions are segregated. The typing judgment for the type-and-effect system is as follows:

$$\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$$

stating that the expression e has type τ, with effects ε, in the typing environment Γ. The effects ε is a finite set of possible effects the evaluation of e can have and consists of read effects rd_ρ, write effects wr_ρ, and allocation effects, al_ρ, for region variables ρ in either the public region context Π or the private region context Λ. Intuitively, public regions are those shared with other threads where as private regions are owned exclusively. The idea is that *only* mutable effects in the public regions are visible to the environment. This is particularly visible in the function types, $\tau \rightarrow_\varepsilon^{\Pi,\Lambda} \tau$, which is annotated with *latent effects* for public and private regions to capture any potential side-effects of evaluating it.

Private regions can be introduced by the masking rule:

$$\frac{\Pi \mid \Lambda, \rho \mid \Gamma \vdash e : \tau, \varepsilon \qquad \rho \notin FRV(\Gamma, \tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon - \rho} \text{TMask}$$

The rule expresses when we can introduce a new private region ρ for the evaluation of an expression e and hide all of e's effects on region ρ. The condition $\rho \notin FRV(\Gamma, \tau)$ ensures that we do not leak any locations of ρ. In earlier work, a similar masking rule has been used for memory-management [76] and for hiding local effects to enable more program-transformations [10, 74].

Private regions can be made public for the duration of a parallel execution by the TPar rule:

$$\frac{\Pi, \Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1 : \tau_1, \varepsilon_1 \qquad \Pi, \Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2 : \tau_2, \varepsilon_2}{\Pi \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma_1, \Gamma_2 \vdash e_1 \| e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2} \text{TPar}$$

The private region context is split in Λ_1, Λ_2 and Λ_3, with Λ_i being the private region context for e_i, and the shared part Λ_3 moved to the public region for both e_1 and e_2.

The stack module in Figure 21 can be given the following type τ_{Stack}:

$$\mathbf{1} \to^{\rho,-}_{\{al_\rho\}} (\mathbf{int} \to^{\rho,-}_{\{wr_\rho, rd_\rho, al_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{\rho,-}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathbf{int})$$

This type expresses that the module will allocate in a public region ρ and return two functions – *push* and *pop*. The type further expresses that *push* is allowed to have read, write and allocate effects in the local state described by ρ and that *pop* can read and write in ρ.

We show in Krogh-Jespersen et al. [42], by the means of LR_{Par}, that the above stack module is contextually equivalent to a module that uses a reference to a pure stack. Intuitively, this holds because their internal data representations are purely local and hidden from clients of the modules.

Theorem 2.1.1. $\rho \mid - \mid - \vdash stack \cong_{ctx} stack_{pure} : \tau_{\text{Stack}}, \{al_\rho\}$

We can further restrict the possible interference from the environment on the stack module by asserting that region ρ should be private, as expressed by the type τ'_{Stack}:

$$1 \to^{-,\rho}_{al_\rho} (\mathbf{int} \to^{-,\rho}_{\{wr_\rho, rd_\rho, al_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathbf{int})$$

Let *stack_nc* be the implementation of *stack* having the **CAS** loop swapped with ordinary assignment. Then we can use our logical relation to prove the following equivalence

Theorem 2.1.2.

$$- \mid \rho \mid - \vdash stack\_nc \cong_{ctx} stack : \tau'_{\mathsf{Stack}}, \{al_\rho\}$$

The above equivalence is valid because the type τ'_{Stack} require the stack modules to be owned exclusively. As a result, operations on the modules will run sequentially, thereby removing the need to use **CAS** for synchronization.

The Parallelization Theorem

Having a static type system can also be used to perform optimizations during the compilation phase. In our setting, we can perform effect-based optimizations based on the effect types. The most interesting effect-based optimization is a parallelization theorem expressing the equivalence of running expressions e_1 and e_2 in parallel and running them sequentially, assuming their effects are suitably disjoint.

Theorem 2.1.3 (Parallelization)**.** *If*

1. $\Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1 : \tau_1, \varepsilon_1$ *and* $\Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2 : \tau_2, \varepsilon_2$
2. $\mathsf{als}\, \varepsilon_1 \cup \mathsf{wrs}\, \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{als}\, \varepsilon_2 \cup \mathsf{wrs}\, \varepsilon_2 \subseteq \Lambda_2$
3. $\mathsf{rds}\, \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ *and* $\mathsf{rds}\, \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then $\cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma_1, \Gamma_2 \vdash e_1 \parallel e_2 \cong_{ctx} (e_1, e_2) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$.

The theorem states that if there exists region contexts, Λ_1, Λ_2 and Λ_3, such that e_i is well-typed having Λ_i and Λ_3 as private and public region contexts (item 1), and if all write and allocation effects of e_i is confined to Λ_i (item 2), and e_i is allowed to read from $\Lambda_i \cup \Lambda_3$ (item 3), then running e_1 and e_2 in parallel is contextually equivalent to running them sequentially. The parallelization theorem is difficult to prove sound when considering a higher-order language with general references and dynamic allocation, having many intricate subtleties.

To show contextual refinement in a concurrent language, one usually shows that, for related heaps h_I and h_S, a step in an implementation $e_I; h_I$ can be simulated by zero or more steps in the specification $e_S; h_S$ producing related heaps h'_I and h'_S. Generally, following the approach of Turon et al. [78], this relational property can be described as a unary Hoare triple by having an exclusive specification thread-reduction resource $j \Rightarrow e$ expressing that the term e is in an evaluation context for a thread informally identified by j:

$$e_I \leq e_S \approx \{j \Rightarrow e_S * heap(h_S)\}$$
$$e_I$$
$$\{v_I. \exists v_S, h'_S.\, j \Rightarrow v_S * heap(h'_S) * e_S; h_S \rightarrow^* v_S; h'_S * \phi(v_I, v_S)\}$$

Here, the post-condition says that if e_I terminates, a reduction from $e_S; h_S \to^* v_S; h'_S$ exists, that the simulation resource is updated to a value v_S (requires exclusive ownership) and that the *heap* resource is updated to the resulting heap h'_S.

For proving relatedness of the ordinary parallel composition $e_1 \| e_2$, we need to be able to split the specification resource $j \Rightarrow e_{1S} \| e_{2S}$ into two separate resources, one for e_{1S} and another for e_{2S}. Then we can pass one to e_{1I} and the other to e_{2I} and they can each reduce their corresponding specification expression, independently of the other. This is possible because e_{1S} and e_{2S} both occur in evaluation contexts.

For the left-to-right case of Theorem 2.1.3, we need to show, that for a reduction step taken in $e_{1I} \| e_{2I}; h_I$ a reduction step must be taken in $(e_{1S}, e_{2S}); h_S$. This may not be possible when taking a step in e_{2S}, unless e_{1S} is fully reduced to a value and e_{2S} is in an evaluation context, thus previous methods for proving parallelization therefore relied on reordering steps taken in e_{2S} while preserving the semantic invariants [9, 14]. For our purposes, the naïve use of $j \Rightarrow v_S$ proves insufficient since we cannot split the resource on a sequential reduction; that would violate having full ownership.

A key contribution in Krogh-Jespersen et al. [42] is a novel technique for proving parallelization that is based on framing.

Conceptually, instead of relating an execution on the left with an execution on the right, the model $\mathrm{LR}_{\mathrm{PAR}}$ relates an execution on the left with all "legal" simulations on the right, that is, all simulations that terminates with related values and ends up in related configurations.

Technically, we use a simulation identifier ζ to keep track of a particular simulation heap $heap_\zeta(h_S)$ and reduction $j \overset{\zeta}{\Rightarrow} v_S$.

To show the parallelization theorem in our model, we suspend the current simulation $j \overset{\zeta}{\Rightarrow} (e_{1S}, e_{2S})$ in configuration $heap_\zeta(h_S)$. Due to the type-and-effect annotation, the heap h_S can be split into: a mutable part h_1 for e_1, a mutable part h_2 for e_2 and an immutable part h_F shared by both. We then construct two new *semi-independent* simulations for e_1 and e_2, $j \overset{\zeta_1}{\Rightarrow} e_{1S}$ and $j \overset{\zeta_2}{\Rightarrow} e_{2S}$, in initial configurations $heap_{\zeta_i}(h_i \uplus h_F)$. When the simulations are finished, they can be reassembled in the original simulation ζ using framing. The formalization of this argument is *not* straight forward and it leverages Iris' facility for capturing sophisticated ownership disciplines.

Related Work

Relational models of type-and-effect systems have been well-studied in increasingly sophisticated sequential programming languages, initiated by the work of Benton *et al.* [8, 10–14, 74]. In this section we only touch on the closest related work and leave the rest for §3.4.

Birkedal et al. [14] showed a relational model for a concurrent language with the same type-and-effect system used in our work. The relational model in [14] was defined as a step-indexed Kripke logical relation. The authors used the model to prove a parallelization theorem similar to ours, informally by proving that the right hand side expression for parallel composition could be delayed and *catch-up* if it was safe to do so, somewhat similar to our notion of simulation. A technical byproduct of that approach meant that they disallowed the delayed computation e_2 to have any allocation effects. In contrast, we build in support for parallelization in the $\mathrm{LR}_{\mathrm{PAR}}$ relation through its notion of multiple simulations. This allowed us to reduce the proof of the parallelization theorem to the essence of why it holds: framing. Additionally, the logical relation presented in [14] only allowed for much more restricted invariants thus it could not be used to prove equivalences such as the ones in 2.1.1 and 2.1.2.

Benton et al. [9] also considered a concurrent language, which, in contrast to the language considered here, only includes first-order store. Technically, this makes the construction of a logical relations model simpler by avoiding the type-world circularity problem. Additionally, their type and effect system does not support dynamic allocation of abstract locations, corresponding to regions in our work, which we support through the masking rule. Conversely, their effect system supports a notion of abstract effects, which means, *e.g.*, that an operation in a data structure module can be considered pure even if it uses effects internally, as long as those effects are not observable from the outside of the module. Benton *et al.* used this facility to show refinement of fine-grained concurrent data structures. Our semantics also supports refinements between fine-grained concurrent data structures, using Iris' support for general invariants.

2.2 Relational Model for Monadic Encapsulation of State

Section §2.1 gave an account of a logical relations model for a language with a type-and-effect system. Another way to characterize effectful computations in functional programming languages is to do it *monadically*. This is done by running the effectful computation in an *encapsulated* environment dictated by the type of the monad and not allowing "impure" values to escape. Since values cannot normally escape the monad, functional languages with support for monads, *e.g.* Haskell, is often considered *pure* languages.

One particular interesting aspect of the type-and-effect system above was the possibility of hiding effects via the masking rule, conceptually allowing values to escape. This is in fact also possible with monads, namely the ST monad introduced by Launchbury and Jones [47]. The ST monad comes equipped with a function $\mathtt{runST} : (\forall \beta, \mathtt{ST}\ \beta\ \tau) \to \tau$ that allows a value to

escape from the monad. runST runs a stateful computation of the monadic type ST β τ and then returns the resulting value of type τ.

The relation between the ST monad and a type-and-effect system should not come as a surprise, as it was already mentioned in [47] that there seems to be a connection between encapsulation using runST and effect masking in type-and-effect systems à la Gifford and Lucassen [26]. This connection was formalized by Semmelroth and Sabry [69], who showed how a language with a simplified type-and-effect system with effect masking can be translated into a language with runST.

In Timany et al. [75], included in Chapter 3, the main technical result is a logical relations model of STLang, a call-by-value, higher-order, functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with runST. The operational semantics of STLang uses a *single global mutable heap*, capturing how the language would be implemented in reality in contrast to earlier work [48, 52]. The main contributions of the work is as follows:

- We present a logical relations model defined using a new approach involving novel predicates for STLang – a language featuring a parallel to Haskell's ST monad.
- We use the logical relation to show that runST provides proper encapsulation of stateful computations, by showing: (i) contextual refinements and equivalences expected to hold for pure languages and (ii) a State-Independence theorem.
- The technical development have been formalized in the Iris implementation, which is a higher-order, concurrent separation logic framework, in Coq, including all proofs of the equations and the State-Independence theorem.

It is the first model that validates the above-mentioned equivalences ((i) and (ii)) for a programming language with a single global heap and in-place destructive updates. The model is described in detail in Figure 46. In the remainder of this section, we discuss the challenges of modeling STLang with runST and the technical innovations required to produce [75].

The ST Monad, STLang and Results Hereof

The ST monad, described in Launchbury and Jones [47] and implemented in Haskell's standard library, is a family of ST β monads satisfying the Kleisli arrow interface:

```
return :: α → ST β α
(>>=)  :: ST β α → (α → ST β α') → ST β α'
```

The monad comes equipped with functions to allocate, read and write to references of type `STRef` β α where α captures the type of the reference cell:

```
newSTRef   :: α → ST β (STRef β α)
readSTRef  :: STRef β α → ST β α
writeSTRef :: STRef β α → α → ST β ()
```

The type variable β, informally, identifies a logical region of the heap to which the function:

```
runST :: (∀ β. ST β α) → α
```

can perform effectful computations on.

STLang is an untyped, call-by-value, higher-order language with constructs similar to the ST monad described above. We use `return` and `bind` for the return and bind (»=) operations of the ST monad, respectively. `ref`(e) creates a new reference, `e` ← `e` writes to a reference and `!e` reads from one.

The operational semantics for STLang is original, so we explain it briefly here. It is a small-step relation, $\rightarrow$, that relates pairs of heaps and expressions and is defined as the closure of a head-step relation $\rightarrow_h$ by evaluation contexts. Interestingly, to reduce effectful computations, the plain reduction has an embedded effectful reduction $\rightsquigarrow$:

$$\frac{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}{\langle h, \texttt{runST}\ \{v\}\rangle \rightarrow_h \langle h', \texttt{runST}\ \{e\}\rangle}$$

Notice that $\rightsquigarrow$ always reduces from a *value*, conceptually, values of type ST are "frozen" computations until run inside `runST`.

Typing judgments, $\Xi \mid \Gamma \vdash e : \tau$, are standard, where Ξ is an environment of type variables, Γ an environment associating types to variables, e is an expression, and τ is a type. With the operational semantics and typing judgments briefly discussed, we can state the State Independence theorem, proven in [75]:

Theorem 2.2.1 (State Independence)**.**

$$\begin{aligned}\cdot \mid x : \mathit{STRef}\ \rho\ \tau' \vdash e : \tau \wedge (\exists h_1, \ell, h_2, v.\ \langle h_1, e[\ell/x]\rangle \rightarrow^* \langle h_2, v\rangle) \implies \\ \forall h_1', \ell'.\ \exists h_2', v'.\ \langle h_1', e[\ell'/x]\rangle \rightarrow^* \langle h_2', v'\rangle \wedge h_1' \subseteq h_2'.\end{aligned}$$

This theorem says that, if the execution of a well-typed expression e terminates, with x substituted by some location, in *some* heap h_1, then e, when x is substituted by *any* location l' in *any* heap h_1', also terminates in some heap h_2'. The heap h_2' is an extension of h_1', *i.e.*, the execution cannot modify h_1'; it can only allocate new state, via encapsulated stateful computations.

Contextual refinement, $\Xi \mid \Gamma \vDash e \preceq_{\text{ctx}} e' : \tau$, is defined for well-typed terms. As usual, e and e' are contextually equivalent, denoted $\Xi \mid \Gamma \vDash e \approx_{\text{ctx}} e' : \tau$, if e contextually refines e' and vice versa.

The contextual refinements and equivalences proven in [75] for pure computations are given in Figure 42 and those for monadic computations are shown in Figure 43. We will touch on the proof of Rec hoisting after shortly presenting the challenges of building a logical relation for STLang. For a full account, we refer to [75].

Challenges of Building a Logical Relations Model for STLang

We mentioned in the section above, that values of type ST $\rho\ \tau$ can be considered as frozen computations, that is, well-typed values that will produce a value τ when run under runST. To that end, we need some logical machinery when stating the value relation, $[\![\Xi \vdash \tau]\!]_\Delta$, for the above type in the logical relations model, where Ξ is an environment of type variables, and Δ is a semantic environment for these type variables, as is usual for languages with parametric polymorphism [65].

Assume that an update modality is defined in Iris, $\Rightarrow P$, that allow resources to be updated (by allocation, modification or deallocation) to resources that satisfy P. Timany et al. [75] defines a *future modality* based on the update modality:

$$\Rightarrow\!\!>\!\!\{n\}\!\equiv\!\!\triangleright P \triangleq (\Rightarrow \triangleright)^n \Rightarrow P$$

which intuitively express that we can update resources to satisfy P, n steps into the future. The future modality can be used to tie the updating of resources together with the operational steps taken, by a predicate referred to as *if-convergent (IC)*, also defined in [75]:

$$\begin{aligned}\mathsf{IC}^\gamma\, e\, \{v.\ Q\} \triangleq \forall h_1, h_2, v, n.\ \langle h_1, e\rangle \rightarrow^n \langle h_2, v\rangle \ast \\ \mathit{heap}_\gamma(h_1) \twoheadrightarrow \Rightarrow\!\!>\!\!\{n\}\!\equiv\!\!\triangleright\ \mathit{heap}_\gamma(h_2) \ast Q\ v\end{aligned}$$

where $\twoheadrightarrow$ is the ordinary magic wand of separation logic. The IC predicate express that for *any* heap h_1, if $\langle h_1, e\rangle$ can reduce to a value v and heap h_2, and we have ownership over logical heap γ with contents h_1, the heap can be updated n steps later to h_2. The IC predicate allows us to reason abstractly and not consider concrete heaps.

The IC predicate can be used to define IC-triples in the same way as weakest pre-condition is used to define Hoare-triples in Iris:

$$\{P\}\, e\, \{v.\ Q\}_\gamma \triangleq \mathit{persistent}(P \twoheadrightarrow \mathsf{IC}^\gamma\, e\, \{v.\ Q\})$$

The quantification over the logical heap, γ, is crucial when giving a semantic type to frozen computations of type ST $\rho\ \tau$, since runST are *pure* expressions

and therefore should yield the same result no matter heap configuration:

$$
\begin{aligned}
&[\![\Xi \vdash \mathsf{ST}\ \rho\ \tau]\!]_\Delta(v, v') \triangleq \forall \gamma_h, \gamma_h', h_1'.\\
&\left\{ heap_{\gamma_h'}(h_1') * \mathsf{region}(\Delta, \rho, \gamma_h, \gamma_h') \right\}\\
&\quad \mathtt{runST}\ \{v\}\\
&\left\{ w.\ \exists h_2', v'.\ \langle h_1', \mathtt{runST}\ \{v'\}\rangle \rightarrow_d^* \langle h_2', v'\rangle * heap_{\gamma_h'}(h_2') * [\![\Xi \vdash \tau]\!]_\Delta(v', \cdot) \right\}_{\gamma_h}
\end{aligned}
$$

Here, region is a predicate tying the logical region ρ to some semantic region with implementation and specification side heaps identified by γ_h and γ_h'. It is worth to reiterate that the segregation of heaps is just a logical division – the operational semantics is defined by means of a global single heap.

Notice, that defining the logical relation in terms of IC-triples, unlike the standard way of giving the expression relation in terms of Hoare-triples, allows one to omit a concrete heap on the implementation side. This approach solves the same problem for the implementation side that the encoding of simulations in Krogh-Jespersen et al. [42] solved for the specification side.

The logical relation defined in [75] is sufficient for validating most of the refinements stated, however, proving Rec hoisting is particularly difficult to prove:

$$\mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{rec}\, f(x) = e_2 \ \preceq_{\mathsf{ctx}}\ \mathtt{rec}\, f(x) = \mathtt{let}\, y = e_1\, \mathtt{in}\, e_2 \ :\ \tau_1 \to \tau_2$$

The number of steps do not align for the steps taken on the implementation and specification side. To show this refinement, a slightly stronger version of the logical relation is required, that force the number of steps taken on both sides to be the same. All this work, including soundness of both logical relations, all refinements and the State-Independence theorem, are proven in Coq.

Related Work

The closest related work to our work in Timany et al. [75] is the original seminal work of Launchbury and Jones [47] and [48], in which the authors discovered that the use of parametric polymorphism in the type for `runST` should still ensure proper encapsulation of effectful computations. However, in Launchbury and Peyton Jones [48], the semantics and parametricity results is denotational and does not use a global mutable heap with in-place update. The authors state in [48, Section 9.1], that proving the remaining part of the language remains pure for an implementation with in-place updates "would necessarily involve some operational semantics.". In Timany et al. [75], we have shown such results for a language with a defined operational semantics with a single global heap with in-place updates.

Moggi and Sabry [52] showed type soundness of calculi with `runST`-like constructs for a call-by-value language and for a lazy language. The results

were shown with respect to an operational semantics in which memory was divided into separate regions: a `runST`-encapsulated computation always started out in an empty heap and the final heap of such a computation was discarded. Timany et al. [75] argue, that such an operational semantics is not realistic for any real-world language implementation. Additionally, the models in [52] are not relational and therefore not suitable for proving relational statements such as the ones shown in Timany et al. [75]. We discuss other related work in §4.7.

2.3 Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems

Relational models are necessary for showing relational properties about programs, as shown the parallelization theorem and rec hoisting above, however, defining unary models to allow reasoning about programs can be sufficiently difficult for some languages. This is true in particular for languages capable of programming distributed systems. Previous work on verification of distributed systems has traditionally focused on verification of protocols of core network components by model-checking, such as SPIN Holzmann [30], TLA+ [46] and Mace [37]. More recently, significant contributions has been made in the field of formal proofs of implementations of challenging protocols, such as two-phase-commit, lease-based key-value stores, Paxos and Raft [28, 50, 62, 70, 81].

One particular issue when verifying distributed systems is the problem of composition. Composing programs from modules that implement protocols is not well-studied by most other verification efforts, with [50, 70] as notable exceptions. One reason for this is that the specification languages of most other verification efforts is based on first order logic, which makes specifying modules and modular reasoning more difficult than it would be for higher-order logics. Additionally, distributed systems composed of individually verified nodes belonging to different protocols, such as a verified load-balancer that balance requests amongst servers, is not well-studied either, because most other works consider a global system, dictated by some state-transition system and thus lack the ability to reason about a node in isolation.

In Krogh-Jespersen et al. [43], included in Chapter 5, we present Aneris, a framework for verifying real-world distributed systems in Iris [38], specifically developed to do *node-local reasoning*, a concept similar to *thread-local reasoning* for concurrent programs, ideal for building verified modules and nodes. In summary, the key contributions of Krogh-Jespersen et al. [43] is as follows:

- AnerisLang, a formalized higher-order functional programming language, with higher-order node-local store, concurrency (threads) and

network sockets, allowing for dynamic creation and binding of sockets to addresses.

- Aneris, a higher-order, separation logic to reason about distributed systems, with support for node-local state and threaded concurrency. Aneris has an adequacy result, which says, that if a system can be bootstrapped in the logic, it is *safe* to run, *i.e.*, it will not crash.
- A simple, novel, approach to guarding network sockets as arbitrary predicates on messages, allowing for asynchronous ownership-transfer between sockets and composition of protocols. Together with a *node-start* rule, similar to *fork* for threads, we obtain the possibility of verifying nodes in isolation. We refer to this as **node-local reasoning**, the basic principle that allows for *modular reasoning* of distributed systems components.
- We use Aneris to verify different interesting examples, including a load-balancer, which is a program that distributes work on multiple servers by the means of threaded concurrency. Additionally, we also verify a module that implements the *two-phase commit* protocol along with a distributed client of the two-phase commit that does replicated logging.

Aneris is the first logic that allows for reasoning about distributed systems with node local state and threaded concurrency, and the first logic to define *node-local* reasoning. Since the logic is built on top of Iris, assertions on state and protocols can use all of the features from Iris, including invariants, monoids and state-transition systems (which is basically just a monoid). In this section we highlight a few of the main principles of Aneris and defer to Chapter 5 for a lengthier discussion.

AnerisLang and its Operational Semantics

Aneris is a framework consisting of a language, AnerisLang, and a logic build on top of Iris, which we refer to as Aneris. AnerisLang is an untyped call-by-value, higher-order concurrent functional language with general references. In addition it has constructs for creating sockets, binding sockets, and sending and receiving messages. The syntax is quite readable and expressible, as shown in the lock server example below (taken from Figure 51):

```
rec lockserver ip p :=
  let lock := ref NONE in
  let skt := socket() in
  Socketbind skt (makeaddress ip p);
  listen skt (rec h msg from :=
    if msg = "LOCK"
    then match !lock with
         NONE => lock ← SOME ();
                 sendto skt "YES" from
       | SOME _ => sendto skt "NO" from
         end
    else lock ← NONE;
         sendto skt "RELEASED" from);
    listen skt h)
```

```
rec listen skt handler :=
  match receivefrom skt with
    SOME m => handler (fst m)
                      (snd m) in
  | NONE => listen skt handler
  end
```

The lock server declares a node-local variable `lock` to keep track of the lock. It then creates and binds a socket `skt` on the given address `ip` and port and starts to continuously listen for incoming messages on the bound socket. When a "LOCK" message arrives [1] and the lock is available, the lock is taken and the server responds "YES". If the lock was already taken, the server responds with "NO". Finally, if the request is not "LOCK", the lock is released and the server responds with "RELEASED".

The semantics of AnerisLang is quite involved since it models sockets in addition to node-local threads and state. Conceptually, a socket is an abstract representation of a handle for a local endpoint of some channel. In Aneris we restrict sockets to use the User Datagram Protocol (UDP), which is *asynchronous*, *connectionless* and *stateless*. In accordance with UDP, Aneris provides no guarantee of delivery or ordering, although we assume duplicate protection, since spatial resources could otherwise potentially be duplicated.

The operational semantics is defined in multiple stages; the first being a node-local, thread-local, small-step head-relation between expressions, heap and allocated sockets shown in Figure 55.

The node-local relation, $\rightarrow_h$, Figure 56, is then lifted to a network-aware stepping relation, tracking the heap and sockets for all nodes, all bound-addresses, all ports in use and all messages sent. The final *distributed-systems* relation reduces by either taking a step in an existing thread on any node or by forking of a new thread.

The simplified inference rules below (see Figure 56 for a full account)

[1] Operationally, messages are pairs of $String \times Address$, but for the lock server example we do not use the second component.

show the semantics for message passing with sockets:

$$\frac{address(z) = \mathsf{Some}\, from \qquad m = (from, to, msg, \textsc{Sent}) \qquad m_{id} \notin dom(M)}{\langle n; \mathsf{sendto}\, z\, msg\, to\rangle, M \rightarrow_h \langle n; \mathsf{length}\ msg\rangle, M[m_{id} \mapsto m]}$$

$$\frac{address(z) = \mathsf{None} \qquad m = (from, to, msg, \textsc{Sent}) \qquad m_{id} \notin dom(M)}{\langle n; \mathsf{sendto}\, z\, msg\, to\rangle, M \rightarrow_h \langle n; \mathsf{length}\ msg\rangle, M[m_{id} \mapsto m]}$$

$$\frac{\begin{array}{c} address(z) = \mathsf{Some}\, a \qquad m_{id} \mapsto m \in M \qquad state(m) = \textsc{Sent} \\ m' = (from(m), a, msg(m), \textsc{Received}) \end{array}}{\langle n; \mathsf{receivefrom}\, z\rangle, M \rightarrow_h \langle n; \mathsf{Some}\,(msg(m), from(m))\rangle, M[m_{id} \mapsto m']}$$

$$\frac{address(z) = \mathsf{Some}\, a \qquad \emptyset = \{m_{id} \mid m_{id} \mapsto (-, a, -, \textsc{Sent}) \in M\}}{\langle n; \mathsf{receivefrom}\, z\rangle, M \rightarrow_h \langle n; \mathsf{None}\rangle, M}$$

One can send a message through a socket by sendto, to which, the message will be added to the message soup M. If the socket is bound, the *from* field of the message will be the address of the bound socket. If the socket is unbound, *from* is some quantified address that is free in the system with an ip-address matching the node. When calling receivefrom there are two possible outcomes. Either the message soup M has messages in Sent state waiting to be received or there are no messages available. If a message is received, the message soup is updated with the state of the message changed to Received.

It is noteworthy that inter-process communication can happen in multiple ways in Aneris. Thread-concurrent programs can communicate through the store but they can also communicate by sending messages through sockets. There is no shared state between nodes thus they can only communicate by message-passing through sockets.

Node-local Reasoning and Protocols

Similarly to *thread-local* reasoning for concurrent separation logic [56], Aneris' logic allows for *node-local* reasoning about programs, *i.e.*, verification of a node in a distributed system is done in isolation with the environment as a frame. This can be seen in the Start-rule by the pre- and post-condition having no explicit assertions on other nodes in the distributed system:

$$\frac{\textsc{Start-Rule}\quad \{P * freePorts(ip, \{p | 0 \le p \le 65536\})\}\, \langle n; e\rangle\, \{x.\ \mathsf{true}\}}{\{P * freeIp(ip)\}\, \langle \mathfrak{S}; \mathsf{start}\ \{n; ip; e\}\rangle\, \{x.\ x = \langle \mathfrak{S}; ()\rangle\}}$$

Here start is the command that launches a new node named n in the distributed system associated with ip-address ip running program e. The predicate $freePorts(ip, P)$ denotes the available ports P for an ip ip, that the program

e can bind sockets on. Note that only the distinguished system node $\mathfrak{S}$ can start new nodes. In Aneris, the execution of the system starts with the execution of $\mathfrak{S}$ as the only node in the distributed system.

In Aneris we associate each socket endpoint (pair of ip-addresses and ports) with a protocol which restricts what can be communicated over that socket. The protocol is an Aneris assertion, $s \Mapsto^{\mathsf{prot}} \Phi$, associating a socket s with a predicate Φ of type $\mathit{Message} \to \mathit{iProp}$. Socket protocols agree on predicates, thus, if we have $s \Mapsto^{\mathsf{prot}} \Phi$ and $s \Mapsto^{\mathsf{prot}} \Psi$, we can conclude that Φ and Ψ is the same protocol.

Aneris supports two kinds of socket endpoints: static socket endpoints and dynamic socket endpoints. This distinction is only at the level of the logic and not the distributed system itself. Static socket endpoints are those which have primordial protocols agreed upon before bootstrapping the system, which makes them ideal for servers. By having primordial protocols and by protocols having to agree on predicates, any node in the system must respect primordial protocols, including the server itself. To track primordial socket endpoints, we use $\overset{f}{\mapsto}(A)$ where A is a set of addresses.

The primordial socket protocol for a lock server can be specified as follows:

$$
\begin{aligned}
\mathit{lock}(m,\phi) &\triangleq \mathit{body}(m) = \text{"}\mathtt{LOCK}\text{"} * \\
&\qquad ((\forall m'.\, \mathit{body}(m') = \text{"}\mathtt{NO}\text{"} \lor \mathit{body}(m') = \text{"}\mathtt{YES}\text{"} * R) \mathrel{-\!\!*} \phi(m')) \\
\mathit{rel}(m,\phi) &\triangleq \mathit{body}(m) = \text{"}\mathtt{RELEASE}\text{"} * R * \\
&\qquad (\forall m'.\, \mathit{body}(m') = \text{"}\mathtt{RELEASED}\text{"} \mathrel{-\!\!*} \phi(m)) \\
\mathit{lock\_si} &\triangleq \lambda m.\, \exists \phi.\, \mathit{from}(m) \Mapsto^{\mathsf{prot}} \phi * (\mathit{lock}(m,\phi) \lor \mathit{rel}(m,\phi))
\end{aligned}
$$

A universally quantified resource describing the lock, R, is transferred to the client if the server responds "YES" and the same resources must be returned when calling "RELEASE". Additionally, the lock protocol also illustrates how primordial servers respond to dynamic bound sockets. The lock server socket must be primordial in practice, however, the lock does not need to know about its clients as long as the clients follow the socket protocol defined by the lock server. As a consequence, a client has to prove that it can receive a reply from the server by proving the resource-aware implication $\mathrel{-\!\!*}$ known as magic wand (expanded upon in 5.4).

A node-local specification for the lock server is as follows:

$$
\begin{aligned}
&\left\{R * (ip,p) \Mapsto^{\mathsf{prot}} \mathit{lock\_si} * \overset{f}{\mapsto} (\{(ip,p)\} \cup A) * \mathit{freePorts}(ip,\{p\})\right\} \\
&\quad \langle n; \mathsf{lockserver}\,() \rangle \\
&\{\mathsf{True}\}
\end{aligned}
$$

There are several interesting observations one can make on the lock server example:

- The lock server can allocate, read and write node-local references but these are hidden in the specification. This is in contrast to [70] which do not have true local references and thus they have to be part of the specification.
- Sockets can be created and bound to specified endpoints. In this example we expect the lock server to be primordial, *i.e.*, the system should agree on a protocol $(ip, p) \Mapsto^{\mathsf{prot}} lock\_si$. Notice as well that there are no channel descriptors or assertions on the socket in the code.
- Without a proper protocol, the lock server fails to provide mutual exclusion since everyone can release the lock. However, with the protocol defined, one can rely on the environment satisfying all stated protocols and as a result no client will try to release without owning the lock.

In Krogh-Jespersen et al. [43] we show two more interesting examples, replicated logging and load-balancing, which we briefly discuss below.

Replicated logging by two-phase commit The two-phase commit protocol (TPC) is a well-studied consensus protocol, however, as mentioned earlier, verifying clients of TPC has received almost no interest.

In Krogh-Jespersen et al. [43] we verify a TPC coordinator and participant module. The TPC module is completely parametric in the event handlers and shape of the messages used for consensus. This allows for different use cases, *e.g.* an auction service or voting scheme.

In [43] we verify an instance of replicated logging as the client of two-phase commit. A central server listens for incoming log messages and initiates rounds of two-phase commit to ask a collection of databases to append the log.

Load balancer A load balancer is crucial for horizontal scaling of a distributed system. A load balancer program forwards request from clients to one of the available servers to which it balances the work load. It then waits for the answer from the server and relays it back to the client.

In order to be able to handle requests from several clients simultaneously, the load balancer can employ concurrency by forking off a new thread for every available server in the system. Each of these threads will then race for incoming requests.

The load balancer is completely modular and can provide load-balancing to an array of services, as long as all of the socket protocols involved do not depend on the sender in any specific way. Since Aneris is the first logic to provide node-local concurrency, no other distributed verification efforts could verify such an example.

Verification effort The total verification effort needed to verify replicated logging with two-phase commit was *only* 1,272 lines in total. Adding load balancing to an existing service and proving adequacy is around 200 lines in total. These results indicate that verifying distributed systems in Aneris is not too demanding.

Related Work

In this section we describe the closest related work to Krogh-Jespersen et al. [43] and defer discussion of other related work to §5.8.

Disel, Sergey et al. [70], is a framework for implementing and verifying distributed systems in Coq. It has a shallowly embedded DSL for writing distributed components that can be verified in by means of a separation logic style Hoare-type theory. Disel achieves compositionality by providing a *frame*-like inference rule, along with two novel logical mechanisms: WithInv for strengthening assumptions by elaborating protocol invariants and *send-hooks* for inter-protocol dependencies. In Aneris, all assertions are stable by definition and quantifiable in such a way that WithInv is not needed. Additionally, Aneris allow for node-local state updates, removing the need for *send-hooks*. Aneris is also equipped with an adequacy result which seems hard to prove in Disel.

One of the examples shown in Sergey et al. [70] is two-phase commit, and a client for logging on top of TPC, quite similar to the example shown in §5.6. However, because of Aneris node-local reasoning principle, Aneris' TPC implementation is easier to compose for clients, compared to the one in [70]. This is a side-effect of having node-local state hidden in the specification of a network module and the quantification of protocols.

IronFleet, Hawblitzel et al. [28], allows for building provably correct distributed systems by a novel combination of TLA-style state-machine refinement with Hoare-logic verification in a layered approach, all embedded in Dafny [49]. Connecting the implementation with the specification is achieved by defining a refinement function and by having the implementation abstractly run the specification by ImplInit and ImplNext. The assertions on imperative programs is stated in first-order predicate logic, in contrast to Aneris' higher-order logic, making it difficult to verify and compose advanced network modules in a distributed system. IronFleet also has support for verifying liveness properties which we do not support in Aneris.

Verdi, Wilcox et al. [81], is a framework for writing and verifying implementations of distributed algorithms in Coq, providing a novel approach to network semantics and fault models. To achieve compositionality, the authors introduced *verified system transformers*, that is, a function that transforms one implementation to another implementation. Generally, [81] has two types of system transformers: transmission transformers, which add network fault toleration to nodes and replication transformers which add node failure tol-

erance to the distributed system. In Aneris, one can easily encode different network protocols by adding sequence numbering to the messages sent, and adding acknowledge responses to received messages, by building a module on top of the defined sockets. Additionally, it is not quite clear how system transformers can be used to verify modules and clients separately, as we do with TPC and replicated logging.

2.4 Verifying a Concurrent Data-Structure from the Dartino Framework

The final manuscript included in this dissertation is a case-study on to how specify and verify an underlying, concurrent scheduler-queue of a real-world virtual machine, Google's Dartino Framework. The Dartino Framework is a managed runtime for the Dart language, specifically designed for high throughput on limited devices, such as IoT devices. The Dartino Framework uses a pool of low-level (hardware) *threads* to run high-level Dart *processes*. Each thread has its own process queue, implemented as a doubly-linked list, which we refer to as a *Dartino Queue*.

Having a Dartino Queue per thread serves to reduce contention, although threads may access the queues of other threads. For instance, a thread with no processes may steal one from another thread. In addition to the usual enqueue and dequeue operations, the data structure allows a specific process to be removed from anywhere in the queue. This allows the scheduler to prioritize certain processes – for instance, to immediately schedule a process that is the recipient of a message.

The case study in Krogh-Jespersen et al. [41], included in Chapter 6, applies Iris to the verification of a Dartino Queue. In effect, the case study demonstrates the practicality and effectiveness of the following:

- Using resources in Iris to reason about dynamic allocation and stealing of processes which may be transferred between queues.
- Using logical atomicity in Iris in concert with resource transfer to verify strong specifications that accurately capture the intention for the real-world code. Having a logically atomic specification allow clients to impose their own invariants on the queue, because it appears as if the operations on the queue take effect at a single (atomic) instant in time.

The case-study has been carried out in collaboration with the Google Dartino Team and all development is fully verified in the Coq implementation of Iris.

A Doubly-Linked List as Concurrent Queue

One particular goal with the Dartino Framework, a virtual-machine for the Dart language written in C++, is to increase the computation throughput of concurrent programs that use message passing for communication. To this end, when one Dart process sends a message to another, the recipient is preferentially scheduled. This means that the Dartino Queue, which represents a process queue in the scheduler, must allow for processes that are not at the head to be removed from the queue, concurrently.

Updating the doubly-linked queue requires multiple updates to references, thus, some sort of locking is needed. The Dartino Queue use the head-pointer as a virtual lock for the queue. When enqueueing or dequeuing, a CAS-loop is used to swing the head-pointer to some sentinel, conceptually informing other threads that the queue is locked. When finished, the head-pointer is moved back which logically releases the lock.

A general-purpose queue data structure based on a linked-list typically allocate a new node when enqueueing, to hold the inserted value. However, for process queues such as the Dartino Queue, nodes are process descriptors, and for performance reason, they exists for the lifetime of the process. Thus, to build a doubly-linked list, the process descriptor holds pointers to the queue the process belongs to and its adjacent processes. This makes specification of the queue more involved because one must handle ownership of process objects carefully, since they may belong to multiple queues during their lifetimes.

In Krogh-Jespersen et al. [41], we have faithfully translated the data-structure from C++ to Iris-ML as shown in Figure 61. Giving a specification to the queue is done in §6.4. The specification is non-trivial and requires a lot of logical machinery to allow for arbitrary process removal without allocating new node-structures and still satisfying the doubly-linked property of the queue.

Atomic Triples

One approach to specifying operations on the Dartino Queue such as enqueue, assuming a predicate `qproc p` asserting ownership of the process descriptor for a process p would be the following high-level Hoare-triple:

$$\begin{array}{l}\{\mathtt{qproc\ p} * \mathtt{queue\ q\ l}\} \\ \quad \mathtt{enqueue(q, p)} \\ \{\mathtt{v.\ v} = () * \mathtt{queue\ q\ (l +\!\!+ [p])}\}\end{array}$$

This specifies that calling enqueue with a valid queue `q` and un-enqueued process `p` will result in the process being appended to the queue. Unfortunately, to use this specification, a thread must have ownership of the queue, which is not useful for a concurrent scheduler where the queue is shared.

An alternative specification would be to wrap the queue in an invariant which allow multiple threads to access the queue simultaneously:

$$\begin{array}{l}\{\mathsf{qproc\ p} * \mathtt{inv\ N}\ (\exists \mathsf{l}.\ \mathsf{queue\ q\ l})\} \\ \quad \mathsf{enqueue(q,p)} \\ \{\mathsf{v}.\ \mathsf{v} = () * \mathtt{inv\ N}\ (\exists \mathsf{l}.\ \mathsf{queue\ q\ l})\}\end{array}$$

The problem with the above specification is that we lose the information that enqueue actually appends the process to the queue. Indeed, an implementation of enqueue could not change the queue at all and be correct with respect to such a specification.

Conceptually, the first specification do not allow any concurrent updates to the queue while the second allow all possible concurrent updates to the queue. The optimal specification would allow the client of the queue to determine exactly which concurrent updates are possible. We can achieve such a specification by viewing the update as *logically atomic* [18]:

$$\begin{array}{l}\mathbb{V}\mathsf{l}.\langle \mathsf{qproc\ p} * \mathsf{queue\ q\ l}\rangle \\ \quad \mathsf{enqueue(q,p)} \\ \quad \langle \mathsf{v}.\ \mathsf{v} = () * \mathsf{queue\ q\ (l +\!\!+ [p])}\rangle\end{array}$$

This specification expresses that the process p is atomically appended to the queue q in the execution of enqueue(q,p). The binding of l, representing the contents of the queue at the atomic point in time, allows the client to arbitrarily update the queue during the execution of enqueue, provided that the precondition holds for *some* l up until the atomic update taking effect. Immediately after the atomic update, the postcondition will hold for the value of l at which the precondition held immediately prior.

In Krogh-Jespersen et al. [41] we give an abstract atomic specification to all Dartino Queue operations and show how a the logical atomic specification can be used by a client.

Part II

Publications and Manuscripts

Chapter 3

A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic

MORTEN KROGH-JESPERSEN, Aarhus University, Denmark
KASPER SVENDSEN, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

In Proceedings of the ACM on Programming Languages (POPL), 2017.

Abstract

Recently we have seen a renewed interest in programming languages that tame the complexity of state and concurrency through refined type systems with more fine-grained control over effects. In addition to simplifying reasoning and eliminating whole classes of bugs, statically tracking effects opens the door to advanced compiler optimizations.

In this paper we present a relational model of a type-and-effect system for a higher-order, concurrent programming language. The model precisely captures the semantic invariants expressed by the effect annotations. We demonstrate that these invariants are strong enough to prove advanced program transformations, including automatic parallelization of expressions with suitably disjoint effects. The model also supports refinement proofs between abstract data type implementations with different internal data representations, including proofs that fine-grained concurrent algorithms refine their coarse-grained counterparts. This is the first model for such an expressive language that supports both effect-based optimizations and data abstraction.

The logical relation is defined in Iris, a state-of-the-art higher-order concurrent separation logic. This greatly simplifies proving well-definedness of the logical relation and provides a powerful logic for reasoning in the model.

3.1 Introduction

Programming with and reasoning about effects in higher-order programs is well-known to be very challenging. Over the years, there have therefore been many proposals of refined type systems for taming and simplifying reasoning about effectful programs. Examples include alias types [71], capability type systems [60], linear type systems [23, 40, 53] Hoare type theory [54], permissions-based type systems [61], type-and-effect systems [11, 12, 26, 51], etc. Lately, we have also witnessed some larger-scale implementation efforts on higher-order programming languages, e.g., the Mezzo programming language [61] and the Rust programming language [67], which employ refined type systems to control the use of state in the presence of concurrency.

In this paper, we provide a logical account of an expressive region-based type-and-effect system for a higher-order concurrent programming language $\lambda_{ref,conc}$ with general references (higher-order store). The type-and-effect system is taken from [14]; it is inspired by Lucassen and Gifford's seminal work [26, 51], but also features a notion of public and private regions, which can be used to limit interference from threads running in parallel. Hence it can be used to express effect-based optimizations, as emphasized for type-and-effect systems for sequential languages by Benton *et al.*, see, *e.g.*, [11, 12]. Effect-based optimizations are examples of so-called "free theorems", *i.e.*, they just depend on the types and effects of the involved expressions, not on the particular expressions involved. The most interesting effect-based optimization is a parallelization theorem expressing the equivalence of running expressions e_1 and e_2 in parallel and running them sequentially, assuming their effects are suitably disjoint. Note that this is a relational property, i.e., the intended invariants of the type-and-effect system are relational in nature. Our logical account of the type-and-effect system thus consists of a logical relations interpretation of the types in a program logic, and we prove that logical relatedness implies contextual equivalence. We show that our logical relations interpretation is strong enough to prove the soundness of effect-based optimizations, in particular the challenging parallelization theorem.

Since the programming language $\lambda_{ref,conc}$ includes higher-order store, it is non-trivial to define a logical relations interpretation of the types, as one is faced with the well-known type-world circularity [4] (see [15] for an overview). Here we factor out this challenge, by using a state-of-the-art program logic, Iris [33], as the logic in which we express the logical relations. Iris has direct support for impredicative invariants, as needed for defining logical relations for general references. Iris also supports reasoning about concurrency; in

particular, it supports a form of rely-guarantee reasoning about shared state. We use this facility to capture invariants of private and public regions. Moreover, we show, using simple synthetic examples, how we can also use the logic to prove that syntactically ill-typed programs obey the semantic invariants enforced by the type system. This is important in practice: both Mezzo and Rust contain facilities for programming with statically ill-typed expressions (Mezzo uses dynamic type checks [61] and Rust allows for including unsafe code in statically typed programs [67]) thus models of type-and-effect systems should preferably support reasoning about combinations of statically ill-typed and statically well-typed programs.

Overview of Challenges and Contributions

The typing judgments of our type-and-effect system take the form

$$\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$$

and express that the term e is of type τ and has effect ε in the typing context Γ mapping variables to types. The additional contexts Π and Λ consist of region variables ρ denoting, respectively, the public regions and the private regions that e may use. Intuitively, public regions are those that other threads may also use, whereas private regions are not subject to interference from other threads. Thus, from a thread-local perspective, the segregation describes an expression's expectations of interference from the environment. The effect ε is a finite set of read rd_ρ, write wr_ρ, or allocation effects, al_ρ, the intuition being that if, *e.g.*, $rd_\rho \in \varepsilon$, then e may read a reference belonging to region ρ.

Effect-based optimizations. Using effect annotations we can express the idea of parallelization mentioned above formally as follows (where $\mathsf{rds}\ \varepsilon$ is the set of regions with read effects in ε and likewise for $\mathsf{wrs}\ \varepsilon$ and $\mathsf{als}\ \varepsilon$):

Theorem 3.1.1 (Parallelization)**.** *If* $\Lambda = \Lambda_1, \Lambda_2, \Lambda_3$ *and*

1. $\Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1 : \tau_1, \varepsilon_1$ *and* $\Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2 : \tau_2, \varepsilon_2$
2. $\mathsf{als}\ \varepsilon_1 \cup \mathsf{wrs}\ \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{als}\ \varepsilon_2 \cup \mathsf{wrs}\ \varepsilon_2 \subseteq \Lambda_2$
3. $\mathsf{rds}\ \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ *and* $\mathsf{rds}\ \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then $\cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma_1, \Gamma_2 \vdash e_1 \parallel e_2 \cong_{ctx} (e_1, e_2) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$.

Here Λ_1 are the private regions of e_1, Λ_2 are the private regions of e_2, and Λ_3 are regions that can be used by both e_1 and e_2. The theorem then says, that if the expressions e_i only write and allocate in their private regions (item 2) and only read in private or shared regions (item 3), then, running e_1 in parallel with e_2 is contextually equivalent with running e_1 and e_2 sequentially, *if* the

$$
\begin{aligned}
&stack_1() = \mathsf{let}\ h = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ (push_1(h), pop_1(h))\\
&push_1(h) = \mathsf{rec}\ loop(n).\mathsf{let}\ v = !h\ \mathsf{in}\\
&\qquad\qquad \mathsf{if}\ \mathsf{CAS}(h, v, \mathsf{inj}_2\ (n, v))\ \mathsf{then}\ ()\ \mathsf{else}\ loop(n)\\
&pop_1(h) = \mathsf{rec}\ loop(\_).\mathsf{let}\ v = !h\ \mathsf{in}\ \mathsf{case}(v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (),\\
&\quad \mathsf{inj}_2\ (n, v') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h, v, v')\ \mathsf{then}\ \mathsf{inj}_2\ n\ \mathsf{else}\ loop())
\end{aligned}
$$

(a) Stack_1

$$
\begin{aligned}
&stack_2() = \mathsf{let}\ h = \mathsf{new}\ (\mathsf{new}\ \mathsf{inj}_1\ ())\ \mathsf{in}\ (push_2(h), pop_2(h))\\
&push_2(h) = \mathsf{rec}\ loop(n).\mathsf{let}\ v = !h\ \mathsf{in}\\
&\qquad\qquad \mathsf{if}\ \mathsf{CAS}(h, v, \mathsf{new}\ \mathsf{inj}_2\ (n, v))\ \mathsf{then}\ ()\ \mathsf{else}\ loop(n)\\
&pop_2(h) = \mathsf{rec}\ loop(\_).\mathsf{let}\ v = !h\ \mathsf{in}\ \mathsf{case}(!v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (),\\
&\quad \mathsf{inj}_2\ (n, v'') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h, v, v'')\ \mathsf{then}\ \mathsf{inj}_2\ n\ \mathsf{else}\ loop())
\end{aligned}
$$

(b) Stack_2

Figure 31: The first stack module, (a) Stack_1, has a single reference to a pure list where (b) Stack_2 uses a reference to a linked list.

context is not allowed to access any locations used by the two expressions (expressed by the fact that $\Lambda_1, \Lambda_2, \Lambda_3$ are all private in the conclusion).

Intuitively, this theorem sounds very plausible, perhaps even quite obvious, but proving it formally was an open problem for more than 25 years [14] and it is still very difficult to prove for higher-order languages with general references, such as ours. Indeed, one of our key contributions is a novel proof technique for proving parallelization. To outline our approach, consider proving the left-to-right approximation of the parallelization theorem (Theorem 3.1.1). Then we, in particular, have to show that any reduction step taken by $e_1 \parallel e_2$ can also be taken by (e_1, e_2). In the case where the expression $e_1 \parallel e_2$ takes a step in the right branch, we cannot yet take the corresponding step in (e_1, e_2), unless e_1 has already reduced to a value. Previous methods for proving parallelization therefore relied on reordering steps taken in e_2 with steps from e_1, while preserving the semantic invariants - resulting in very difficult proofs [14] or trace-based arguments [9], which are not known to scale to programming languages with general references and dynamic allocation.

Our new technique is instead based on framing. We suspend the reduction on the right hand side temporarily, and first disentangle the reduction of $e_1 \parallel e_2$ into two semi-independent ("semi" because they can read from shared regions) reductions for e_1 and e_2 respectively, which can then be reassembled into a reduction for (e_1, e_2) using framing. The disentanglement and the re-

assembling qua framing, of course, depends on the effect annotations, and our formal argument leverages Iris' facility for capturing sophisticated ownership disciplines. We present a more detailed description and the formal argument in Section 3.3.

Data abstraction and local state. The $\lambda_{ref,conc}$ language supports hiding of local state using closures. Hiding can be used to implement abstract data types (ADTs) that manipulate an internal data representation, which can only be accessed through the provided operations. Relating ADT implementations that use different internal data representations is well-studied in the setting of ML-like type systems (see, *e.g.*, [3, 79] and the references therein); effect tracking adds several interesting dimensions.

In the ML setting the type system imposes no constraints on local state when relating ADT implementations. This is not the case in our setting. To illustrate, consider the following counter implemented using local state:

$$e_{\text{count}} \triangleq \mathsf{let}\ x = \mathsf{new}\ 0\ \mathsf{in}\ \mathsf{rec}\ inc().\mathsf{let}\ y = !x\ \mathsf{in} \\ \mathsf{if}\ \mathsf{CAS}(x, y, y+1)\ \mathsf{then}\ y\ \mathsf{else}\ inc()$$

e_{count} allocates a local reference x and returns a function that try to increment x inside a loop, until it succeeds, and returns the old value. To allow for concurrent access, the function uses a compare-and-set operation (CAS), which atomically sets the value of x to $y+1$ if the value of x is equal to y and returns **true** or **false** depending on the result. The counter has the following type:

$$\rho \mid - \mid - \vdash e_{\text{count}} : 1 \to^{\rho,-}_{\{rd_\rho, wr_\rho\}} \mathsf{int}, \{al_\rho\}$$

The type is a function type, which is annotated with a latent effect, expressing that the returned function may read and write in the public region ρ. To prove soundness of effect-based transformations, it is, of course, crucial that the semantic model also enforces the semantic invariants expressed by the effect annotations on local state. Otherwise, if our semantic model would allow us to forget about the effects on the local reference x, then we would be able to show, using a semantic version of Theorem 3.1.1, that $\mathsf{let}\ g = e_{\text{count}}\ \mathsf{in}\ g() \| g()$ is contextually equivalent to $\mathsf{let}\ g = e_{\text{count}}\ \mathsf{in}\ (g(), g())$, which is not the case (the first expression may evaluate to $(1,0)$, while the second always evaluates to $(0,1)$).

We can use the type-and-effect system to limit interference from the environment on the internal state of ADTs, when relating ADTs. For example, consider the two stack modules listed in Figure 31. The left stack module, Stack_1, uses a single reference to a pure functional list whereas the right module, Stack_2, uses a linked list representation. Both stack implementations use a CAS operation to ensure that they function correctly in the presence of concurrent interference. The implementations (*i.e.*, $stack_1$ and $stack_2$) can be

given the following type τ_{Stack}:

$$\mathbf{1} \to^{\rho,-}_{\{al_\rho\}} (\mathsf{int} \to^{\rho,-}_{\{wr_\rho, rd_\rho, al_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{\rho,-}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})$$

This type expresses that each module will allocate in region ρ and return two functions *push* and *pop*. The type further expresses that *push* is allowed to have read, write and allocate effects on the local state described by ρ and that *pop* is allowed to read and write.

Intuitively, the two implementations are equivalent at this type, because their internal data representations are purely local and hidden from clients of the modules. Indeed, we can use our logical relation to prove:

Theorem 3.1.2. $\rho \mid - \mid - \vdash stack_1 \cong_{ctx} stack_2 : \tau_{\mathsf{Stack}}, \{al_\rho\}$

Now, if we restrict possible interference from the environment by making the ρ region private, as expressed by the type τ'_{Stack} (ρ is now private on the latent effects, since it comes *after* the comma):

$$1 \to^{-,\rho}_{al_\rho} (\mathsf{int} \to^{-,\rho}_{\{wr_\rho, rd_\rho, al_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})$$

then the two implementations are still contextually equivalent at this type.

Moreover, for this type, we can also prove that we can safely omit the CAS operation from the stack implementations (intuitively, because there is no possible concurrent interference). Thus, writing $stack\_nc_i$ for the implementation of $stack_i$ without a CAS loop, we can use our logical relation to prove the following equivalences.

Theorem 3.1.3.

$$- \mid \rho \mid - \vdash stack\_nc_1 \cong_{ctx} stack_1 : \tau'_{\mathsf{Stack}}, \{al_\rho\}$$
$$\textit{and} \quad - \mid \rho \mid - \vdash stack\_nc_2 \cong_{ctx} stack_2 : \tau'_{\mathsf{Stack}}, \{al_\rho\}$$

Our proofs of data abstraction, detailed in §3.3 and §II, leverage Iris's facility for expressing invariants on local state. As pointed out in [14] the logical relation in *loc.cit.* could not be used to prove equivalences such as this one, since the logical relation there only allowed for much more restricted invariants.

Ill-typed terms. Here is a simple example of a statically ill-typed expression which nevertheless satisfies the semantic invariants enforced by the type system:

$$e \triangleq x := (); x := \mathsf{true}$$

This expression first assigns the unit value to a boolean reference, and then assigns **true** to it.

This expression is not statically typable, due to the assignment of unit to a boolean reference. However, if the boolean reference is *private* then the rest of the program is not allowed to observe the ill-typed intermediate value and will thus never observe that the typing discipline has been broken. It is thus perfectly safe to use the untypable term e *as if* it had the following type:

$$- \mid \rho \mid x : \mathbf{ref}_{\rho}\, \mathsf{B} \vdash e : \mathbf{1}, \{rd_{\rho}, wr_{\rho}\}$$

Our logical relations model allows us also to reason about such statically ill-typed terms and, *e.g.*, prove that e is equivalent to a statically well-typed expression which only assigns **true** to x.

Summary of Contributions In summary, the contributions of this paper are:

- We show how to interpret types of a region-based type-and-effect system for a concurrent higher-order imperative programming language with higher-order store as logical relations in the state-of-the-art program logic Iris.
- We use the interpretation to prove soundness of effect-based optimizations. In particular, we prove the soundness of the parallelization theorem. Our parallelization theorem is a strengthening of the one in [14] and for our proof we use a novel proof technique, based on framing. The resulting proof is arguably a lot clearer and more abstract than the one in [14], thanks to the use of the logical features of Iris.
- We use the interpretation to prove contextual equivalence of fine-grained concurrent data structures that use local state to hide internal data representations. Our examples could not be proved with the logical relation in [14].
- We show how the logic may be used to prove that syntactically ill-typed expressions obey the semantic properties enforced by the type system.
- We demonstrate that the logic allows us to give a modular definition of the logical relation and explain the relation by breaking it down into more manageable parts.

Outline We begin by formally defining the syntax and semantics of $\lambda_{ref,conc}$, the type-and-effect system, and contextual equivalence in §3.2. In §3.3 we turn our attention to logical relations for $\lambda_{ref,conc}$. We present our logical relation in four stages, starting from a unary relation that characterizes type inhabitance and ending with a binary relation for reasoning about contextual equivalence that supports advanced effect-based optimizations, each building on the previous relation. We conclude and discuss related and future work in §3.4.

3.2 $\lambda_{ref,conc}$ with Types, Regions and Effects

In this section we present the operational semantics and the type-and-effect system for $\lambda_{ref,conc}$, a call-by-value language with general references and concurrency primitives $\|$ and CAS (compare-and-set).

Syntax and Operational Semantics of $\lambda_{ref,conc}$

The syntax of $\lambda_{ref,conc}$ is shown in Figure 32 and the operational semantics is summarized in Figure 33. We assume given denumerably infinite sets of variables Var, ranged over by x, y, f, and locations Loc, ranged over by l. We use v to range over the set of values, Val, and e to range over the set of expressions, Exp. Note that expressions do not include types. We use B, true, false and $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ as shorthands for booleans and branching encoded using sums.

$$
\begin{aligned}
v &::= () \mid n \mid (v, v) \mid \mathsf{inj}_i\ v \mid \mathsf{rec}\ f(x).e \mid x \mid l \\
e &::= v \mid e = e \mid e\ e \mid (e, e) \mid \mathsf{prj}_i\ e \mid \mathsf{inj}_i\ e \mid e + e \mid \mathsf{new}\ e \mid !e \mid e := e \\
&\mid \ \mathsf{CAS}(e, e, e) \mid e \| e \mid \mathsf{case}(e, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e)
\end{aligned}
$$

Figure 32: Syntax of $\lambda_{ref,conc}$.

The operational semantics is defined by a small-step relation between configurations consisting of a heap and an expression. Heaps h are finite partial maps from locations to values. The semantics is defined in terms of evaluation contexts, $K \in \text{ECtx}$. We use $K[e]$ to denote the expression obtained by plugging e into the context K and $e[v/x]$ to denote capture-avoiding substitution of value v for variable x in expression e.

Types and Effects for $\lambda_{ref,conc}$

The set of types is defined by the following grammar:

$$\text{Type}\ \tau ::= \mathsf{1} \mid \mathsf{int} \mid \mathsf{ref}_\rho\ \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau \to^{\Pi,\Lambda}_{\varepsilon} \tau$$

Π and Λ are finite sets of region variables, taken from a denumerably infinite set RegVar ranged over by ρ. We use comma to denote disjoint union of sets of region variables. An atomic effect on a region ρ is either a read effect, rd_ρ, a write effect, wr_ρ, or an allocation effect, al_ρ. An effect ε is a finite set of atomic effects. Typing judgments take the form $\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$. An excerpt of the typing rules are shown in Figure 34. All typing rules can be found in §II.

Regions can be introduced by the masking rule (TMask). The masking rule expresses when we can introduce a new private region ρ for the evaluation

Evaluation Contexts

$$\begin{aligned} K ::= {} & [\,] \mid K = e \mid v = K \mid K\ e \mid v\ K \mid (K, e) \mid (v, K) \mid \mathsf{prj}_i\ K \mid K + e \mid v + K \\ \mid {} & \mathsf{inj}_i\ K \mid \mathsf{case}(K, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \mid \mathsf{new}\ K \mid !K \mid K := e \mid v := K \\ \mid {} & K \| e \mid e \| K \mid \mathsf{CAS}(K, e, e) \mid \mathsf{CAS}(v, K, e) \mid \mathsf{CAS}(v, v, K) \end{aligned}$$

Pure reduction $\boxed{e \overset{\text{pure}}{\longrightarrow} e'}$

$$v_1 \| v_2 \overset{\text{pure}}{\longrightarrow} (v_1, v_2)$$

Reduction $\boxed{h; e \to h'; e'}$

$$\begin{aligned} h; e &\to h; e' && \text{if } e \overset{\text{pure}}{\longrightarrow} e' \\ h; \mathsf{new}\ v &\to h \uplus [l \mapsto v]; l \\ h; !l &\to h; v && \text{if } h(l) = v \\ h[l \mapsto -]; l := v &\to h[l \mapsto v]; () \\ h; \mathsf{CAS}(l, v_o, v_n) &\to h; \mathsf{false} && \text{if } h(l) \neq v_o \\ h[l \mapsto v_o]; \mathsf{CAS}(l, v_o, v_n) &\to h[l \mapsto v_n]; \mathsf{true} \\ h; K[e] &\to h'; K[e'] && \text{if } h; e \to h'; e' \end{aligned}$$

Figure 33: Operational semantics of $\lambda_{ref,conc}$. Remaining pure reductions are standard (see Chapter II).

of an expression e and hide all of e's effects on region ρ. The condition $\rho \notin FRV(\Gamma, \tau)$ ensures that we do not leak any locations of ρ and hence, from the perspective of e, region ρ is private. The masking rule has been used to do memory-management [76] and to hide local effects to enable more program-transformations [10, 74].

Since the masking rule allows us to hide local state effects, a pure operation is not necessarily deterministic in our setting. For instance, the following code-snippet which non-deterministically returns **true** or **false** can be typed as a pure expression:

$$- \vdash \mathsf{let}\ x = \mathsf{new}\ \mathsf{true}\ \mathsf{in}\ (x := \mathsf{true} \| x := \mathsf{false}); !x : \mathsf{B}, \emptyset$$

Contextual Equivalence for $\lambda_{ref,conc}$

We take contextual equivalence as our basic notion of equivalence. Contextual equivalence relates two expressions if no suitably typed context can distinguish them. For a concurrent language such as $\lambda_{ref,conc}$ we have to choose whether there simply has to exist an indistinguishable reduction

$$\frac{(x:\tau)\in\Gamma}{\Pi \mid \Lambda \mid \Gamma \vdash x:\tau,\emptyset} \qquad \frac{}{\Xi \vdash () : \mathbf{1},\emptyset} \qquad \frac{\Xi \vdash e_1:\tau,\varepsilon_1 \qquad \Xi \vdash e_2:\tau,\varepsilon_2 \qquad eq_{type}(\tau)}{\Xi \vdash e_1 = e_2 : \mathbf{B},\varepsilon_1\cup\varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma, f:\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2, x:\tau_1 \vdash e:\tau_2,\varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{rec}\ f(x).e : \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2,\emptyset}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1:\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2,\varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2:\tau_1,\varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e_1\ e_2 : \tau_2,\varepsilon\cup\varepsilon_1\cup\varepsilon_2} \qquad \frac{\Xi \vdash e:\mathbf{ref}_\rho\ \tau,\varepsilon}{\Xi \vdash !e:\tau,\varepsilon\cup\left\{rd_\rho\right\}}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e:\tau,\varepsilon \qquad \rho\in\Pi,\Lambda}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{new}\ e:\mathbf{ref}_\rho\ \tau,\varepsilon\cup\left\{al_\rho\right\}} \qquad \frac{\Xi \vdash e_1:\mathbf{ref}_\rho\ \tau,\varepsilon_1 \qquad \Xi \vdash e_2:\tau,\varepsilon_2}{\Xi \vdash e_1 := e_2 : \mathbf{1},\varepsilon_1\cup\varepsilon_2\cup\left\{wr_\rho\right\}}$$

$$\frac{\Pi \mid \Lambda,\rho \mid \Gamma \vdash e:\tau,\varepsilon \qquad \rho\notin FRV(\Gamma,\tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e:\tau,\varepsilon-\rho}\ \textsc{TMask} \qquad \frac{}{eq_{type}(\mathbf{1})}$$

$$\frac{\Pi,\Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1:\tau_1,\varepsilon_1 \qquad \Pi,\Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2:\tau_2,\varepsilon_2}{\Pi \mid \Lambda_1,\Lambda_2,\Lambda_3 \mid \Gamma_1,\Gamma_2 \vdash e_1 \| e_2 : \tau_1\times\tau_2,\varepsilon_1\cup\varepsilon_2}$$

$$\frac{\Xi \vdash e_1:\mathbf{ref}_\rho\ \tau,\varepsilon_1 \qquad \Xi \vdash e_2:\tau,\varepsilon_2 \qquad \Xi \vdash e_3:\tau,\varepsilon_3 \qquad eq_{type}(\tau)}{\Xi \vdash \mathbf{CAS}(e_1,e_2,e_3):\mathbf{B},\varepsilon_1\cup\varepsilon_2\cup\varepsilon_3\cup\left\{wr_\rho,rd_\rho\right\}}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e:\tau_1,\varepsilon_1 \qquad \Pi,\Lambda \vdash \tau_1\le\tau_2 \qquad \varepsilon_1\subseteq\varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e:\tau_2,\varepsilon_2}$$

$$\frac{eq_{type}(\tau) \qquad eq_{type}(\sigma) \qquad op\in\{+,\times\}}{eq_{type}(\tau\ op\ \sigma)}$$

$$\frac{FRV(\tau)\in\Pi}{\Pi \vdash \tau\le\tau} \qquad \frac{\Pi \vdash \tau_1\le\tau_1' \qquad \Pi \vdash \tau_2\le\tau_2'}{\Pi \vdash \tau_1\times\tau_2\le\tau_1'\times\tau_2'}$$

$$\frac{\Pi \vdash \tau_1'\le\tau_1 \qquad \Pi \vdash \tau_2\le\tau_2' \qquad \varepsilon_1\subseteq\varepsilon_2 \qquad \Pi_1\subseteq\Pi_2 \qquad \Lambda_1\subseteq\Lambda_2}{\Pi \vdash \tau_1\to^{\Pi_1,\Lambda_1}_{\varepsilon_1}\tau_2 \le \tau_1'\to^{\Pi_2,\Lambda_2}_{\varepsilon_2}\tau_2'}$$

Figure 34: Excerpt of typing and sub-typing inference rules. We write $FV(e)$ and $FRV(e)$ for the sets of free program variables and region variables respectively. For all typing judgments $\Pi \mid \Lambda \mid \Gamma \vdash e:\tau,\varepsilon$ we implicitly assume that $FRV(\Gamma,\tau,\varepsilon)\in\Pi\cup\Lambda$. The equality type predicate, eq_{type}, defines the types we may test for equality. We use Ξ as shorthand for $\Pi \mid \Lambda \mid \Gamma$.

(may-equivalence) or whether all possible reductions must be indistinguishable (must-equivalence). In this paper we study may-equivalence and may-approximation, as defined below.

Definition 3.2.1. $\Pi \mid \Lambda \mid \Gamma \vdash e_1 \leq_{ctx} e_2 : \tau, \varepsilon$ *iff for all contexts* C, *values* v, *and heaps* h_1 *such that* $C : (\Pi \mid \Lambda \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (- \mid - \mid - \vdash \mathbf{B}, \emptyset)$ *and* $[]; C[e_1] \to^* h_1; v$ *there exists a heap* h_2 *such that* $[]; C[e_2] \to^* h_2; v$.

The $C : (\Pi \mid \Lambda \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (\Pi' \mid \Lambda' \mid \Gamma' \vdash \tau', \varepsilon')$ relation expresses that the context C takes a term e of the former type to a term of the latter type; the definition is standard and relegated to §II. Note that e_1 and e_2 are not required to be well-typed in the definition above. Contextual equivalence $\Pi \mid \Lambda \mid \Gamma \vdash e_1 \cong_{ctx} e_2 : \tau, \varepsilon$ is then defined as contextual approximation in both directions.

3.3 A Logical Relation for $\lambda_{ref,conc}$

In this section we present a logical relation for $\lambda_{ref,conc}$. To aid exposition we present the logical relation in four steps. We start by defining a unary logical relation for a simplified type system without regions and effects. This allows us to focus on the use of Iris as a meta-language for logical relations and provide a gentle introduction to Iris. We then extend the unary relation to the full type system with regions and effects, focusing on how effects are translated into abstract descriptions of possible interference. These unary logical relations characterize type inhabitance, which suffices for establishing type soundness, but not for proving equivalences. As the third step we naively extend the unary relation for the full type system to a binary relation, focusing on how to express a binary relation in a unary program logic. This yields a logical relation that is sound with respect to contextual approximation and suffices for proving equivalences of many concrete examples, but not advanced effect-based equivalences such as parallelization. For the fourth and final relation, we refine the third relation further, to support reasoning using multiple simulations. This final relation validates parallelization and is also sound with respect to contextual approximation.

This staged presentation also highlights the modularity of using Iris as a meta-language for logical relations. In particular, each step builds naturally on the previous, only requiring small changes or additions between each relation.

Unary Relation for $\lambda_{ref,conc}$ Without Effects

We begin by defining a unary logical relation for $\lambda_{ref,conc}$ with a standard ML-like type system without regions and effects. The goal is to define a unary relation, $\mathrm{LR}_{\mathrm{ML}}$ that characterizes type inhabitance semantically and is sound with respect to the syntactic typing rules. More precisely, we wish to define

$$\begin{aligned}
\kappa &::= 1 \mid \mathsf{Exp} \mid \mathsf{Val} \mid \mathsf{Name} \mid \mathsf{Prop} \mid \mathsf{Monoid} \mid \kappa \times \kappa \mid \cdots \\
t, \varphi, \iota, P, \mathcal{M} &::= x \mid \lambda x : \kappa. t \mid \varphi\ t \mid (t, t) \mid \pi_1(t) \mid \pi_2(t) \mid t =_\kappa t \mid t \mapsto t \mid \bot \mid \top \\
&\mid\ P \wedge P \mid P \vee P \mid P \Rightarrow P \mid \forall x : \kappa. P \mid \exists x : \kappa. P \mid P * P \\
&\mid\ P \mathrel{-\!\!*} P \mid \Box P \mid \triangleright P \mid \boxed{P}^{\iota} \mid \{P\}\ e\ \{v.\ Q\}_{\mathcal{M}} \mid P\ {}^{\mathcal{M}}\!\Rightarrow^{\mathcal{M}} Q \mid \cdots
\end{aligned}$$

Figure 35: Excerpt of Iris syntax.

two unary relations, a value relation, $[\![\tau]\!]$, that characterizes values of type τ and an expression relation, $\mathcal{E}[\![\tau]\!]$, that characterizes expressions that either diverge or evaluate to values of type τ.

For ground types the definition of $[\![\tau]\!]$ is obvious: it is the values of the given type τ. The main difficulty arises when defining the interpretation of reference types. The idea is to take a location l to be an inhabitant of type $\mathsf{ref}\ \tau$ if location l contains a value of type τ in the current heap. $\lambda_{ref,conc}$ is a concurrent language and the context is free to update the heap as it sees fit. However, the context must preserve typing and we can thus think of $[\![\mathsf{ref}\ \tau]\!](l)$ as expressing an *invariant* that l must always contain a value v of the semantic type $[\![\tau]\!]$ in the heap. To formalize this we introduce our meta-language, Iris.

Iris and invariants. Iris is a generic framework for constructing higher-order separation logics. For the purposes of this paper we present one particular instance of this framework for the $\lambda_{ref,conc}$ language and we refer to this instance simply as Iris.

Figure 35 contains an excerpt of the Iris syntax. Iris is a higher-order logic over a simply-typed term language. The set of Iris types, ranged over by κ, includes a type of $\lambda_{ref,conc}$ expressions Exp and values Val, a type of propositions, Prop, and is closed under products and function spaces. Iris includes the usual connectives ($\bot, \top, \wedge, \vee, \Rightarrow, \forall, \exists, *, \mathrel{-\!\!*}, =_\kappa$) and proof rules of higher-order separation logic. Iris extends this with a few new primitives, which we explain below.

Iris makes no distinction between assertions and specifications. Specifications are simply treated as special assertions that do not express ownership of any state. This is captured by the always modality, $\Box P$, which expresses that P holds and does not assert ownership of any state. Since $\Box P$ does not assert any ownership, it can be freely duplicated ($\Box P \iff \Box P * \Box P$). We therefore call assertions of the form $\Box P$ *persistent*.

One of the main features of Iris is *invariants* for reasoning about shared state. The pure assertion $\boxed{P}^{\iota}$ asserts the existence of an invariant with the name ι that owns a resource satisfying the assertion P. Resources owned by invariants are shared by every thread and can be accessed freely by atomic operations, provided the invariant is preserved. For atomic operations we can

thus *open* an invariant and take local ownership of the resource owned by the invariant for the duration of the operation, provided we transfer back a resource that satisfies the invariant assertion after the operation. In Iris this is captured formally by *view-shifts*. A view-shift, written $P \Rrightarrow Q$ expresses that it is possible to transform a resource satisfying P into a resource satisfying Q, without changing the underlying physical state. To reason about opening of invariants, view-shifts are further annotated with *invariant masks* indicating which invariants are required to hold before and after the view-shift. In the view-shift $P \ {}^{\mathcal{M}_1}\!\Rrightarrow^{\mathcal{M}_2} Q$, $\mathcal{M}_1$ and $\mathcal{M}_2$ are invariant masks (sets of invariant names) required to hold before and after the view-shift respectively. The invariant masks ensure that we do not open an invariant twice (which would not be sound in general). Opening and closing of invariants is captured by the two following view-shift axioms:

$$\frac{}{\boxed{P}^{\iota} \ {}^{\{\iota\}}\!\Rrightarrow^{\emptyset} \triangleright P}\ \textsc{InvOpen} \qquad \frac{}{\boxed{P}^{\iota} * \triangleright P \ {}^{\emptyset}\!\Rrightarrow^{\{\iota\}} \top}\ \textsc{InvClose}$$

The InvOpen rule allows us to take ownership of P upon opening the invariant ι, while the InvClose rule requires us to relinquish ownership of P to close the invariant ι. In both rules, the resource P is guarded by a modality, $\triangleright$, which we explain shortly.

To apply these view-shifts to open an invariant for the duration of an atomic operation, such as reading ($!e$), writing ($x := e$) or allocating ($\mathsf{new}\ e$), Iris features the following atomic rule-of-consequence.

$$\frac{P_1 \ {}^{\mathcal{M} \uplus \mathcal{M}'}\!\Rrightarrow^{\mathcal{M}} P_2 \qquad \{P_2\}\ e\ \{Q_2\}_{\mathcal{M}} \qquad \forall v.\, Q_2(v) \ {}^{\mathcal{M}}\!\Rrightarrow^{\mathcal{M} \uplus \mathcal{M}'} Q_1(v)}{\{P_1\}\ e\ \{Q_1\}_{\mathcal{M} \uplus \mathcal{M}'}}\ e \text{ atomic}\ \textsc{ACsq}$$

Iris triples, $\{P\}\ e\ \{Q\}_{\mathcal{M}}$, are also annotated with an invariant mask, $\mathcal{M}$, indicating which invariants are required to hold before, during and after the execution of e. The atomic rule-of-consequence allows us to change this mask to open invariants for the duration of an atomic expression e. View-shifts include implication ($\Box(p \Rightarrow q) \vdash p \Rrightarrow q$) and we can thus recover the usual rule-of-consequence from ACsq.

The "later" modality, $\triangleright$, is used to express that a property is only required to hold after one step of execution. It is used in connection with invariants because an Iris invariant may contain *any* predicate P, including one referring to the invariant itself. To ensure this is well-defined, Iris uses a form of guarded recursion, an abstract version of step-indexing — where $\triangleright$ is used to *guard* the resource in the invariant. Since the later modality expresses that a predicate holds after one step of execution, we can remove a $\triangleright$ modality from a precondition whenever our program makes an operational step. This is captured by the frame rule for atomic expressions:

$$\frac{\{P\}\ e\ \{Q\}}{\{P * \triangleright R\}\ e\ \{v.\ Q(v) * R\}}\ e \text{ atomic}\ \textsc{AFrame}$$

For many assertions it is also possible to remove laters without an operational step. We call these assertions *timeless* as they are independent of the number of steps left. For timeless assertions P we can view-shift away laters: $\triangleright P \Rrightarrow P$. Timeless assertions are closed under the connectives and quantifiers of first-order separation logic, but crucially does not include invariant assertions $\boxed{P}^{\iota}$, as the steps are precisely needed to model potentially self-referential invariants. With the exception of the reference invariants we use, all the invariants used throughout this article are timeless.

While the invariant names ι on invariant assertions, view shifts and Hoare triples are important for soundness, they are not important for understanding our encodings of logical relations in Iris. We have therefore chosen to elide all invariant names in the article, and refer interested readers to Chapter II, where everything is fully annotated.

Logical relation. We now have enough logical machinery to define the first unary logical relation in Iris. The full definition of $\mathrm{LR}_{\mathrm{ML}}$ is given in Figure 36.

The value relation, $[\![\tau]\!]$, is defined by induction on τ and defines an Iris assertion of type $\mathsf{Val} \to \mathsf{Prop}$. The expression relation, $\mathcal{E}$, is defined independently of the value relation and takes an arbitrary value predicate and extends it to expressions. It has the following type in Iris:

$$\mathcal{E} : (\mathsf{Val} \to \mathsf{Prop}) \to (\mathsf{Exp} \to \mathsf{Prop})$$

As already mentioned, for ground types, $[\![\tau]\!]$ is simply the set of values of the given type τ. The definition for arrow-types follows the usual idea of related arguments to related values, with the added wrinkle that we only require the argument to be related later. This suffices since applying a function takes a step in the operational semantics. The always modality in the value relation for arrow types is there to ensure the value relation is pure, which allows us to duplicate the resource that witnesses that a value is well-typed. It is needed in the arrow-case as implication does not preserve purity in general. For space reasons we omit the cases for products and sum types and refer the reader to §II.

Finally, for reference types, $\mathsf{ref}\ \tau$, the value relation is the set of locations l such that there exists an invariant that owns the location l and contains a value v in $[\![\tau]\!]$. Resources owned by invariants are shared, which allow all concurrently executing threads to freely update references, provided they respect the typing of the reference. This type of invariant can be seen as a particularly simple instance of rely / guarantee reasoning, where the rely and the guarantee are the same: namely, to preserve the invariant. A large part of the challenge throughout the rest of this article boils down to refining this invariant to limit possible interference from the environment, based on the region and effect system.

The expression relation $\mathcal{E}(\phi)$ extends a value predicate ϕ to expressions e by requiring that, if e terminates, then it terminates with a value satisfying

$$\llbracket \mathbf{1} \rrbracket \triangleq \lambda x.\ x = () \qquad \llbracket \mathbf{int} \rrbracket \triangleq \lambda x.\ x \in \mathbb{N}$$
$$\textsc{Ref}(x, \phi) \triangleq \exists v.\ x \mapsto v * \phi(v) \qquad \llbracket \mathbf{ref}\ \tau \rrbracket \triangleq \lambda x.\ \boxed{\textsc{Ref}(x, \llbracket \tau \rrbracket)}^{\textsc{Rf}(x)}$$
$$\llbracket \tau_1 \to \tau_2 \rrbracket \triangleq \lambda x.\ \Box \forall y.\ (\triangleright \llbracket \tau_1 \rrbracket(y)) \Rightarrow \mathcal{E}(\llbracket \tau_2 \rrbracket)(x\ y)$$
$$\mathcal{E}(\phi)(e) \triangleq \{\top\}\, e\, \{v.\ \phi(v)\}_\top$$

Logical relatedness

$$\overline{x : \tau} \models_{\text{ML}} e : \tau \triangleq \vdash_{\text{Iris}} \forall \overline{x'}.\, \overline{\llbracket \tau \rrbracket}(\overline{x'}) \implies \mathcal{E}(\llbracket \tau \rrbracket)(e[\overline{x'}/\overline{x}])$$

Figure 36: LR_{ML}: Unary rel. for $\lambda_{ref,conc}$ sans effect-types.

ϕ. Finally, $\Gamma \models_{\text{ML}} e : \tau$ extends this to open expressions, by closing under all substitutions. This semantic typing judgment is sound with respect to the usual typing rules, in the sense that for any well-typed term $\Gamma \vdash e : \tau$, the Iris assertion $\Gamma \models_{\text{ML}} e : \tau$ is provable in Iris.

Lemma 3.3.1 (Soundness)**.** *If* $\Gamma \vdash e : \tau$ *then* $\vdash_{\text{Iris}} \Gamma \models_{\text{ML}} e : \tau$.

We note in passing that this logical relation shows the power of using Iris as a meta-language for defining logical relations: Usually, to define logical relations for a language with general references, one would need to index semantic types by worlds containing semantic types for allocated locations and the worlds and semantic types would have to be recursively defined [4, 15]; here this is all taken care of by Iris' built-in general logical facility for defining and working with invariants.

Unary Relation for $\lambda_{ref,conc}$ with Effects

In this section we extend the unary relation from the previous section to the full type system with regions and effects. Note that simply extending the relation from the previous section to the full type system by ignoring all region and effect annotations already yields a relation that is sound with respect to the full type system. However, this is needlessly conservative and by interpreting region and effect annotations as restricting interference, we obtain a more precise semantic typing relation that is also sound.

The idea is to use the distinction between public and private regions to limit interference from the context, and the effect annotations to limit the effects of the given expression. We can encode this in Iris using tokens indexed by a region r corresponding to each type of effect: $[\textsc{Rd}]^\pi_r$, $[\textsc{Wr}]^\pi_r$, $[\textsc{Al}]^\pi_r$. Each token is intended to grant permission to perform the corresponding effect on region r and, depending on the fractional permission $\pi \in \{\pi \in \mathbb{Q} \mid 0 < \pi \leq 1\}$,

prevent the context from performing the given effect. These tokens must satisfy the following properties (and likewise for Wr,Al):

$$[\text{Rd}]^1_r * [\text{Rd}]^\pi_r \Rightarrow \bot \qquad\qquad [\text{Rd}]^{\pi_1+\pi_2}_r \Leftrightarrow [\text{Rd}]^{\pi_1}_r * [\text{Rd}]^{\pi_2}_r \tag{3.1}$$

expressing that the full permission ($\pi = 1$) really means exclusive ownership of the token and that these tokens can be split and recombined arbitrarily. In Iris we can define such tokens using *ghost state*. Below we give a brief introduction to ghost state in Iris; for a more thorough treatment, we refer the reader to [33].

Iris and ghost state. Ghost resources provide a modular way of reasoning about *knowledge* and *rights* to modify some shared state. Ghost state is modeled using partial commutative monoids in Iris. Formally, these partial commutative monoids are presented as total commutative monoids with a distinguished zero element, $\bot$. The assertion $\boxed{m : M}^\gamma$ asserts ownership of a ghost resource $m \in |M|$ of the monoid instance γ. Separating conjunction on ghost state is simply the lifting of the underlying monoid composition:

$$\boxed{m_1 : M}^\gamma * \boxed{m_2 : M}^\gamma \Leftrightarrow \boxed{m_1 \cdot_M m_2}^\gamma \tag{3.2}$$

The zero-element represents an ill-defined resource and thus cannot be owned: $\boxed{\bot : M}^\gamma \Leftrightarrow \textsf{false}$.

Tokens are a degenerate form of ghost state, consisting only of rights. The Frac monoid, defined below, allows us to define an effect token for a single region. The carrier is rationals between 0 and 1, with addition as composition and 0 as the unit (we typically omit the explicit zero element from the definition of the monoid carrier and composition):

$$\text{Frac} = [0,1] \cap \mathbb{Q} \qquad q \cdot q' = q + q', \quad \text{if } q + q' \leq 1$$

The idea is that 0 represents no ownership, 1 exclusive ownership and anything rational in the interval $(0,1)$ non-exclusive ownership.

To define effect tokens for arbitrary regions, we also need the partial finite function monoid, $\text{FpFun}(X, M)$, with unit ε being the empty map and whose carrier is functions f from a set X into the non-zero elements of a monoid M, such that the set $\{x \in X \mid f(x) \neq \varepsilon\}$ is finite. Composition on $\text{FpFun}(X, M)$ is defined point-wise, but is only defined if all point-wise compositions are well-defined:

$$(f \cdot g)(x) \triangleq f(x) \cdot g(x) \qquad\qquad \text{if } f(x) \cdot g(x) \neq \bot \text{ for all } x \in X$$

Effect tokens can now be defined as follows and proven to satisfy property (3.1). The proof is an easy consequence of (3.2) and the definition of the monoid.

$$[\text{X}]^\pi_r \triangleq \boxed{[r \mapsto \pi] : \text{FpFun}(\mathcal{RN}, \text{Frac})}^X, \; X \in \{\text{Rd}, \text{Wr}, \text{Al}\}$$

Ghost state is a purely logical construct and is updated using view-shifts rather than assignments. To update a ghost resource we must ensure that our update is consistent with all ghost resources potentially owned by the environment. This is captured by the GhostUpd rule given below:

$$\frac{\text{GhostUpd}\quad \forall m_f.\, m \cdot m_f \neq \bot \Rrightarrow \exists m' \in M'.\, m' \cdot m_f \neq \bot}{\boxed{m : M}^{\gamma} \Rrightarrow \exists m' \in M'.\, \boxed{m' : M}^{\gamma}}$$

To update a ghost resource m to some element $m' \in M'$, we have to show that doing so preserves all possible frames m_f composable with the resource m.

We can instantiate the finite partial functions monoid with locations and values to obtain the standard monoid of heaps used in separation logic. To define a monoid on values, we extend it with a unit element and a composition operator that is only defined if one of the two elements is unit.

$$\textsc{Heap} \triangleq \textsc{FpFun}(\textsc{Loc}, \textsc{Val} + \{\varepsilon\})$$

The ghost resource $\boxed{[l \mapsto v] : \textsc{Heap}}^{\gamma}$ asserts the exclusive right to modify location l in ghost heap γ and that location l currently contains the value v (here we use $[l \mapsto v]$ for the function that maps l to v and every other argument to ε). Using the GhostUpd rule, we can update ghost locations we own and allocate new ghost locations:

$$\boxed{[l \mapsto v] : \textsc{Heap}}^{\gamma} \Rrightarrow \boxed{[l \mapsto v'] : \textsc{Heap}}^{\gamma} \tag{3.3}$$

$$\boxed{[] : \textsc{Heap}}^{\gamma} \Rrightarrow \exists l.\, \boxed{[l \mapsto v] : \textsc{Heap}}^{\gamma} \tag{3.4}$$

Throughout the rest of the article we will need many ghost resources, including the Heap monoid. We will introduce them by explaining the properties we expect them to satisfy. Naturally, we must define monoids for all of these resources and prove that the desired properties hold. All of these definitions and proofs can be found in §II.

Encoding effects using ghost state. Now that we have seen how to define and work with ghost state in Iris, we proceed with how to encode effects using ghost state.

A read-effect on a private region translates into exclusive ownership of the corresponding read token, while a read-effect on a public region translates into ownership of the corresponding read token with an arbitrary fractional permission π (and likewise for write and allocation effects).

The intended meaning of these tokens is enforced through the interplay between two invariants: a new *region* invariant, $\textsc{Reg}(r)$, linking references with their corresponding region, and an updated reference invariant, $\textsc{Ref}(r, \phi, x)$, indexed by a region identifier r and the reference's semantic type ϕ. Before we define these formally, we review some properties that should hold. If we

own part of the read token for a region r then the context knows we might read references belonging to this region and must ensure that their values are well-typed. This is captured by the following property (where all free variables are universally quantified):

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)} \vdash \{[\textsc{Rd}]^{\pi}_{r}\}\ !x\ \{y.\,[\textsc{Rd}]^{\pi}_{r} * \phi(y)\} \tag{3.5}$$

Preservation of well-typedness is expressed by $\phi(y)$ in the post-condition. If we own part of the write token for a region r then we should be allowed to write any well-typed value to a reference belonging to region r:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)}, \phi(v) \vdash \{[\textsc{Wr}]^{\pi}_{r}\}\ x := v\ \{y.\,[\textsc{Wr}]^{\pi}_{r}\} \tag{3.6}$$

Likewise, if we own any part of the allocation token for a region r we should be allowed to allocate a new reference and associate it with region r:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \phi(v) \vdash \{[\textsc{Al}]^{\pi}_{r}\}\ \mathsf{new}\ v\ \{y.\ [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Ref}(r,\phi,y)}^{\textsc{Rf}(y)}\}$$

Those three properties were fairly uneventful; the interesting properties deal with exclusive ownership of effect tokens.

Exclusive read effect. If we own the read token for region r exclusively, then the context cannot rely on references in region r containing well-typed values. If we additionally own a write token for region r, then we should be allowed to assign arbitrary values to references belonging to region r, provided we restore them with well-typed values before returning the exclusive read token. To capture this formally, we introduce two new tokens, $[\textsc{Rd}(x)]_r$ and $[\textsc{NoRd}(x)]_r$, which express that if location x belongs to region r then it contains a well-typed value and may contain a value that is not well-typed, respectively. If we own the read token on a region r exclusively, then the following property allows us to exchange it for tokens that force all locations belonging to region r to contain well-typed values.

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash [\textsc{Rd}]^{1}_{r} \Longleftrightarrow \circledast_{x}[\textsc{Rd}(x)]_{r} \tag{3.7}$$

By giving up the token that expresses that a location contains a well-typed value, we can assign an arbitrary value to the location. If we later assign a well-typed value, we can recover the token witnessing the well-typedness of the location. This is captured by the following two properties.

$$\begin{aligned}&\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)} \\ &\quad\vdash \{[\textsc{Wr}]^{\pi}_{r} * [\textsc{Rd}(x)]_{r}\}\ x := v\ \{y.\ [\textsc{Wr}]^{\pi}_{r} * [\textsc{NoRd}(x)]_{r}\}\end{aligned} \tag{3.8}$$

$$\begin{aligned}&\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)}, \phi(v) \\ &\quad\vdash \{[\textsc{Wr}]^{\pi}_{r} * [\textsc{NoRd}(x)]_{r}\}\ x := v\ \{y.\ [\textsc{Wr}]^{\pi}_{r} * [\textsc{Rd}(x)]_{r}\}\end{aligned} \tag{3.9}$$

Exclusive write effect. If we own the full write token, $[\textsc{Wr}]^1_r$, then the context should not be allowed to modify references belonging to region r. Again, we capture this property by introducing new tokens $[\textsc{Wr}(x)]_r$ and $x \overset{\pi}{\hookrightarrow}_r v$. Both tokens express that if location x belongs to region r, then we own the exclusive right to update it; the latter token further asserts that the current value is v. As before, we can trade ownership of a per-region write token for region r for all per-location write tokens for region r:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash [\textsc{Wr}]^1_r \Longleftrightarrow \circledast_x [\textsc{Wr}(x)]_r \tag{3.10}$$

Given ownership of a per-location write token for a location x belonging to region r, we can trade the token for a *points-to proxy* for x with fractional permission $\frac{1}{2}$:

$$\boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)} \vdash [\textsc{Wr}(x)]_r \Longleftrightarrow \exists v.\, x \overset{\frac{1}{2}}{\hookrightarrow}_r v \tag{3.11}$$

This points-to proxy satisfies similar properties as the standard separation logic points-to: If we own the points-to proxy $x \overset{\pi}{\hookrightarrow}_r v$ and read location x, we will read the value v:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash \{x \overset{\pi}{\hookrightarrow}_r v\}\ !x\ \{y.\, y = v * x \overset{\pi}{\hookrightarrow}_r v\} \tag{3.12}$$

If we own half of the points-to proxy for a location x we can also use it to assign a well-typed value to x:

$$\begin{gathered}\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)}, \\ \phi(v_2) \vdash \{x \overset{\frac{1}{2}}{\hookrightarrow}_r v_1\}\ x := v_2\ \{y.\, x \overset{\frac{1}{2}}{\hookrightarrow}_r v_2\}\end{gathered} \tag{3.13}$$

Exclusive ownership of the per-region write token thus allows us to reason about the exact value of all references belonging to the region.

Exclusive allocation effect. Exclusive ownership of a per-region allocation token allows us to lock the domain of the heap associated with the given region. By trading our exclusive per-region allocation token, we can take ownership of a new token, $[\textsc{Al}(h)]^\pi_r$, that witnesses the domain of the heap associated with the given region:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash [\textsc{Al}]^1_r \Longleftrightarrow \exists h.\, [\textsc{Al}(h)]^{\frac{1}{2}}_r \tag{3.14}$$

As usual, we use fractional permissions to share the $[\textsc{Al}(h)]^\pi_r$ token. Given fractional ownership of two parts of the $[\textsc{Al}(h)]^\pi_r$ token, the domains of the two heaps must agree:

$$[\textsc{Al}(h_1)]^{\pi_1}_r * [\textsc{Al}(h_2)]^{\pi_2}_r \Rightarrow [\textsc{Al}(h_1)]^{\pi_1}_r * [\textsc{Al}(h_2)]^{\pi_2}_r * \mathrm{dom}(h_1) = \mathrm{dom}(h_2)$$

Exclusive ownership ($\pi = 1$) is required to update the domain of the heap. It is possible to update the heap without exclusive access, as long as the domain is preserved:

$$[\textsc{Al}(h)]^1_r \Rrightarrow [\textsc{Al}(h')]^1_r$$
$$[\textsc{Al}(h)]^\pi_r * \mathrm{dom}(h) = \mathrm{dom}(h') \Rrightarrow [\textsc{Al}(h')]^\pi_r$$

Logical relation. The $\mathrm{LR}_{\textsc{Eff}}$ logical relation, including the new region invariant and updated reference invariant is defined in Figure 37. Changes to existing predicates are in green. The value relation, $[\![\tau]\!]^M$, is now indexed by an injective mapping M from region variables to Iris invariant names. This mapping allows us to model that the same region variable might refer to different regions in the case where a region ρ is created after hiding a region with the same name ρ. Likewise, the expression relation, $\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)$, is also indexed by the region mapping M, in addition to the region contexts Π, Λ and the effect typing ε.

To explain the relation, let us start with the reference invariant $\textsc{Ref}(r, \phi, x)$. Note first that the reference invariant no longer owns the underlying physical location (i.e., $x \mapsto v$). Instead it owns a proxy $x \overset{\frac{1}{2}}{\hookrightarrow}_r v$. The *effs* predicate encodes the meaning of the per-location read and write tokens. It allows us to exchange a write token $[\textsc{Wr}(x)]_r$ for a proxy that describes the current value of the location (property (3.11)) and track when the location contains a well-typed value (properties (3.8) and (3.9)).

The region invariant $\textsc{Reg}(r)$ consists of two resources, a token resource $toks(r)$ that ties all the per-region tokens together with the per-location tokens, and the $locs(r)$ resource that ties together the points-to proxies with the physical state. The $toks(r)$ resource allows us to exchange an exclusive per-region read or write token for all the corresponding per-location read or write tokens (properties (3.7) and (3.10)). It also enforces that if we only own a fraction of the per-region read or write token then the region invariant must own all per-location read and write tokens for the given region. This ensures that the location must contain a well-typed value and that we are allowed to update it, respectively (properties (3.5) and (3.6)).

The local points-to proxies for a region r are tied to the physical state using the global counter-part $rheap(h, r)$ resource in $locs(r)$. The local points-to proxy always agrees with the global heap proxy:

$$rheap(h, r) * x \overset{\pi}{\hookrightarrow}_r v \Rrightarrow rheap(h, r) * x \overset{\pi}{\hookrightarrow}_r v * h(x) = v$$

To update a points-to proxy thus requires ownership of both the corresponding global heap proxy and exclusive ownership of the local points-to proxy:

$$rheap(h, r) * x \overset{1}{\hookrightarrow}_r v \Rrightarrow rheap(h[x \mapsto v'], r) * x \overset{1}{\hookrightarrow}_r v'$$

New predicates

$$
\begin{aligned}
\mathit{effs}(r,\phi,x,v) &\triangleq ([\textsc{Wr}(x)]_r \lor x \overset{\frac{1}{2}}{\hookrightarrow}_r v) * ([\textsc{Rd}(x)]_r \lor (\phi(v) * [\textsc{NoRd}(x)]_r)) \\
\textsc{Reg}(r) &\triangleq \mathit{locs}(r) * \mathit{toks}(r) \\
\mathit{locs}(r) &\triangleq \exists h.\, \mathit{rheap}(h,r) * \mathit{alloc}(h,r) * \circledast_{(l,v)\in h} l \mapsto v * \\
&\qquad \circledast_{\{x \mid x\in(\mathit{Loc}\setminus \mathrm{dom}(h))\}} [\textsc{NoRd}(x)]_r \\
\mathit{toks}(r) &\triangleq ([\textsc{Rd}]^1_r \lor \circledast_{x\in\mathit{Loc}} [\textsc{Rd}(x)]_r) * ([\textsc{Wr}]^1_r \lor \circledast_{x\in\mathit{Loc}} [\textsc{Wr}(x)]_r) \\
\mathit{alloc}(h,r) &\triangleq ([\textsc{Al}]^1_r * [\textsc{Al}(h)]^{\frac{1}{2}}_r) \lor [\textsc{Al}(h)]^1_r \\
P_{toks}(\rho,r,\pi,\varepsilon) &\triangleq (\rho \notin \mathsf{rds}\ \varepsilon \lor [\textsc{Rd}]^\pi_r) * (\rho \notin \mathsf{wrs}\ \varepsilon \lor [\textsc{Wr}]^\pi_r) * \\
&\qquad (\rho \notin \mathsf{als}\ \varepsilon \lor [\textsc{Al}]^\pi_r) \\
P_{reg}(R,g,\varepsilon,M) &\triangleq \circledast_{\rho\in R} P_{toks}(\rho, M(\rho), g(\rho), \varepsilon) * \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))} \\
P^{\Pi,\Lambda}(g,\varepsilon,M) &\triangleq P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M) * P_{reg}(\Pi, g, \varepsilon, M)
\end{aligned}
$$

Changes to previous definitions

$$
\begin{aligned}
\textsc{Ref}(r,\phi,x) &\triangleq \exists v.\ x \overset{\frac{1}{2}}{\hookrightarrow}_r v * \mathit{effs}(r,\phi,x,v) \\
[\![\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M &\triangleq \lambda x.\ \Box \forall y.\ (\triangleright\, y \in [\![\tau_1]\!]^M) \Rightarrow \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(x\ y) \\
[\![\mathsf{ref}_\rho\ \tau]\!]^M &\triangleq \lambda x.\ \boxed{\textsc{Ref}(M(\rho), [\![\tau]\!]^M, x)}^{\textsc{Rf}(x)} \\
\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)(e) &\triangleq \forall g.\ \{P^{\Pi,\Lambda}(g,\varepsilon,M)\}\, e\, \{v.\ \phi(v) * P^{\Pi,\Lambda}(g,\varepsilon,M)\}_\top
\end{aligned}
$$

Logical relatedness

$$
\Pi \mid \Lambda \mid \overline{x:\tau} \models_{\textsc{Eff}} e : \tau, \varepsilon \triangleq \vdash_{\textsc{Iris}} \forall M. \forall \overline{x'}.\, \overline{[\![\tau]\!]^M}(\overline{x'}) \implies \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau]\!]^M)(e[\overline{x'}/\overline{x}])
$$

Figure 37: $\mathrm{LR}_{\textsc{Eff}}$: Unary rel. for $\lambda_{ref,conc}$ with effect-types.

The $\mathit{locs}(r)$ resource asserts ownership of physical points-to resources for each location and value in the global points-to proxy for region r. Since these are tied together with the local points-to proxies, the local points-to proxies must agree with the underlying physical state (properties (3.12) and (3.13)). $\mathit{locs}(r)$ also asserts ownership of all the $[\textsc{NoRd}(x)]_r$ resources for locations x not belonging to the region.

The reason for introducing the indirection of proxies, is to allow reasoning about the set of locations belonging to a region, to interpret allocation effects. This is captured by the $\mathit{alloc}(h,r)$ resource, which allows clients to trade the exclusive allocation token for a lock on the set of locations belonging to the region (property (3.14)).

Finally, P_{reg} specifies how the effect annotation translates into ownership of the corresponding effect tokens. Effects in private regions yield exclusive ownership, while effects in public regions yield non-exclusive ownership.

Example: type violating update. To illustrate how we can use this stronger semantic typing judgment to semantically type-check code that is not syntactically well-typed, recall the previous mentioned type-violating example.

$$- \mid \rho \mid x : \mathsf{ref}_\rho\ \mathsf{B} \models_{\text{Eff}} x := (); x := \mathsf{true} : \mathbf{1}, \{rd_\rho, wr_\rho\}$$

The read and write effect on the private region ρ translates into exclusive ownership of the read and write token. Using properties (3.7), (3.8) and (3.9) we can thus easily verify that the example is semantically well-typed.

Context: $\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r, [\![\mathsf{B}]\!]^M, x)}^{\textsc{Rf}(x)}$

$\left\{[\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\right\} \Rrightarrow \left\{[\textsc{Wr}]^1_r * \circledast_{y \in Loc}[\textsc{Rd}(y)]_r\right\}$

$x := ()$

$\left\{[\textsc{Wr}]^1_r * [\textsc{NoRd}(x)]_r * \circledast_{y \in Loc \setminus \{x\}}[\textsc{Rd}(y)]_r\right\}$

$x := \mathsf{true}$

$\left\{[\textsc{Wr}]^1_r * \circledast_{y \in Loc}[\textsc{Rd}(y)]_r\right\} \Rrightarrow \left\{[\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\right\}$

Binary Relation for $\lambda_{ref,conc}$ with Effects

Previously we looked at unary relations for semantically characterizing type inhabitance. Now, we switch to binary relations intended to imply contextual approximation.

We define two families of binary relations, $[\![\tau]\!]^M$ and $\mathcal{E}([\![\tau]\!]^M)$, that characterize contextual approximation on values and expressions of type τ, respectively. Generalizing the value relation to contextual approximation is fairly straightforward: on ground types it is simply the identity relation on the values of the given type; for arrow types it relates functions that map related arguments to related expressions, and for reference types it relates two locations if they contain related values.

The expression relation is more interesting: intuitively it should express that e_I approximates e_S, if any step that e_I can make can be simulated by zero or more steps of e_S. We think of e_I as an "implementation" and of e_S as a "specification". We follow the approach of Turon *et al.* [78] and capture this relational property as a unary Hoare triple on e_I by requiring the triple to update ghost resources that force the execution of e_S. The idea is to introduce a ghost resource $j \Rightarrow_S e$ that expresses that the expression e is in an evaluation context on the "specification" side and the exclusive right to reduce this

New predicates

$$\textsc{Spec}(h_0, e_0) \triangleq \exists h, e.\, heap_S(h) * mctx(e) * (h_0; e_0 \to^* h; e)$$

Changes to previous definitions

$$\begin{aligned}
\textsc{Ref}(r, \phi, x) &\triangleq \exists v.\ x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v_S * effs(r, \phi, x, v) \\
effs(r, \phi, x, v) &\triangleq ([\textsc{Wr}(x)]_r \vee (x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v_S)) * \\
&\quad ([\textsc{Rd}(x)]_r \vee (\phi(v_I, v_S) * [\textsc{NoRd}(x)]_r)) \\
locs(r) &\triangleq \exists h.\, rheap_I(h_I, r) * rheap_S(h_S, r) * alloc(h, r) \\
&\quad \circledast_{(l,v)\in h_I} l \mapsto_I v * \circledast_{(l,v)\in h_S} l \mapsto_S v \\
&\quad \circledast_{x\in Loc\setminus \mathrm{dom}(h_I)\times(Loc\setminus\mathrm{dom}(h_S))} [\textsc{NoRd}(x)]_r \\
tokens(r) &\triangleq ([\textsc{Wr}]^1_r \vee \circledast_{x\in Loc^2} [\textsc{Wr}(x)]_r) * ([\textsc{Rd}]^1_r \vee \circledast_{x\in Loc^2} [\textsc{Rd}(x)]_r) \\
alloc(h, r) &\triangleq ([\textsc{Al}]^1_r * [\textsc{Al}(h_I, h_S)]^{\frac{1}{2}}_r) \vee [\textsc{Al}(h_I, h_S)]^1_r \\
[\![\mathbf{1}]\!]^M &\triangleq \lambda x.\ x_I = x_S = () \\
[\![\mathbf{int}]\!]^M &\triangleq \lambda x.\ x_I, x_S \in \mathbb{N} \wedge x_I = x_S \\
[\![\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M &\triangleq \lambda x.\ \Box \forall y.\ (\triangleright [\![\tau_1]\!]^M)(y_I, y_S) \Rrightarrow \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(x_I\ y_I, x_S\ y_S) \\
[\![\mathbf{ref}_\rho\ \tau]\!]^M &\triangleq \lambda x.\ \boxed{\textsc{Ref}(M(\rho), [\![\tau]\!]^M, x)}^{\textsc{Rf}(x)} \\
\mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}(\phi)(e_I, e_S) &\triangleq \forall g, j, h_0, e_0.\ \boxed{\textsc{Spec}(h_0, e_0)} \vdash \\
&\quad \{j \Rrightarrow_S e_S * P^{\Pi,\Lambda}(g, \varepsilon, M)\} \\
&\quad\ e_I \\
&\quad \{v_I.\ \exists v_S.\ j \Rrightarrow_S v_S * \phi(v_I, v_S) * P^{\Pi,\Lambda}(g, \varepsilon, M)\}_\top
\end{aligned}$$

Logical relatedness

$$\begin{aligned}
&\Pi \mid \Lambda \mid \overline{x : \tau} \models_{\textsc{Bin}} e_1 \le_{log} e_2 : \tau, \varepsilon \triangleq \\
&\quad \vdash_{\textsc{Iris}} \forall M.\, \forall \overline{x}.\, \overline{[\![\tau]\!]^M}(\overline{x}) \Rrightarrow \mathcal{E}^{\Pi,\Lambda}_{\varepsilon,M}([\![\tau]\!]^M)(e_1[\overline{x_I}/\overline{x}], e_2[\overline{x_S}/\overline{x}])
\end{aligned}$$

Figure 38: $\mathrm{LR}_{\textsc{Bin}}$: Binary rel. for $\lambda_{ref,conc}$ with effect-types.

expression. With this ghost resource we can express a simulation between an implementation e_I and a specification e_S as follows:

$$e_I \le e_S \approx \{j \Rrightarrow_S e_S\}\ e_I\ \{v_I.\, \exists v_S.\, j \Rrightarrow_S v_S * \phi(v_I, v_S)\}$$

By requiring e_I to update the ghost resource from e_S to a value v_S, we are forced to show that we can reduce e_S, which appears in an evaluation con-

text on the specification side, to the value v_S. We refer to $j \Rightarrow_S e$ as a local expression resource, as it allows us to reason locally about reductions on sub-expressions of the full specification program.

The generalization to expressions in evaluation contexts is necessary to prove that the relation is a congruence. In particular, to prove the following congruence property:

$$e_{1I} \leq e_{1S} \wedge e_{2I} \leq e_{2S} \Rightarrow e_{1I} \| e_{2I} \leq e_{1S} \| e_{2S}$$

We need to be able to split the local expression resource $j \Rightarrow_S e_{1S} \| e_{2S}$ into two separate resources, one for e_{1S} and another for e_{2S}. Then we can pass one to e_{1I} and the other to e_{2I} and they can each reduce their corresponding expression on the right, independently of the other. This is possible because e_{1S} and e_{2S} both occur in evaluation contexts. This is also the reason why local expression resources $j \Rightarrow_S e$ are indexed. The j serves as a logical "thread identifier", allowing us to distinguish different local expression resources.

Specification resources. To formalize this idea, we need a number of ghost resources. In addition to the local expression resource, $j \Rightarrow_S e$ we also need a global expression resource, $mctx(e)$, for reasoning about the full specification program. Naturally, the global expression resource and all local expression resources must agree on the specification program, so splitting a local expression resource requires ownership of both. The following lemma allows us to introduce and eliminate a local expression resource for an expression that occurs in an evaluation context inside another local expression resources:

$$j \Rightarrow_S \kappa[e_1] * mctx(e) \Longleftrightarrow \exists i.\, j \Rightarrow_S \kappa[i] * i \Rightarrow_S e_1 * mctx(e) \tag{3.15}$$

This is achieved by introducing a new logical thread identifier i for the new local expression resource for e_1 and replacing e_1 with i in the original local expression resource. Here κ is an evaluation context extended to expressions that may contain logical thread identifiers. By applying the above property twice, we can split a local expression resource for a parallel composition into two:

$$j \Rightarrow_S e_1 \| e_2 * mctx(e) \Longleftrightarrow \exists i_1, i_2.\, j \Rightarrow_S i_1 \| i_2 * i_1 \Rightarrow_S e_1 * i_2 \Rightarrow_S e_2 * mctx(e)$$

Since all local specification expressions are in an evaluation context of the global specification expression, any reduction of a local specification expression can be extended to the global specification expression:

$$\begin{aligned} &j \Rightarrow_S e_1 * mctx(e) * (h; e_1 \to h'; e_1') \Rrightarrow \\ &\quad \exists e'.\, j \Rightarrow_S e_1' * mctx(e') * (h; e \to h'; e') \end{aligned} \tag{3.16}$$

In the case where there exists just one local expression resource that contains no free logical thread identifiers, then the local expression should agree with

the global expression. To formalize this, we treat the thread identifier 0 as the "root" local expression:

$$0 \Rightarrow_S e_1 * mctx(e_2) * FA(e_1) = \emptyset \;\Rightarrow\; 0 \Rightarrow_S e_1 * mctx(e_2) * e_1 = e_2 \tag{3.17}$$

where $FA(e)$ is the set of free logical thread identifiers in e. These local and global expression resources are definable in Iris and we refer the reader to §II for detailed definitions

We need another two ghost resources, $heap_S(h)$ and $l \mapsto_S v$, for reasoning about specification heaps. This is in fact the Heap monoid we have seen before, with some additional structure. The $heap_S(h)$ resource asserts global ownership of the full specification heap h, while $l \mapsto_S v$ asserts local ownership of a single location l, respectively. We require that the global heap agrees with the local heap resources:

$$heap_S(h) * l \mapsto_S v \Rightarrow heap_S(h) * l \mapsto_S v * h(l) = v \tag{3.18}$$

Updating a location l requires both local ownership of l and the global heap resource and allocation requires ownership of the global heap resource, both lifted from updating and allocating ghost locations seen before in (3.3) and (3.4).

$$heap_S(h) * l \mapsto_S v \Rrightarrow heap_S(h[l \mapsto v']) * l \mapsto_S v' \tag{3.19}$$

$$heap_S(h) * l \notin \mathrm{dom}(h) \Rrightarrow heap_S(h[l \mapsto v]) * l \mapsto_S v \tag{3.20}$$

With these resources in hand, we can now formally define a simulation as a Hoare triple. We define a specification invariant that asserts ownership of the global specification heap and expression. Additionally, it also requires that there exists a reduction from some initial configuration $h_0; e_0$ to the current global specification heap and expression:

$$\textsc{Spec}(h_0, e_0) \triangleq \exists h, e.\, heap_S(h) * mctx(e) * (h_0; e_0 \rightarrow^* h; e)$$

By requiring the Hoare triple to update the local expression e_S of a to a value v_S, we thus force it to show the existence of a reduction:

$$\boxed{\textsc{Spec}(h_0, e_0)}^{\iota} \vdash \{j \Rightarrow_S e_S\}\ e_I\ \{v_I.\, \exists v_S.\, j \Rightarrow_S v_S\}$$

The only way to update the local expression resource $j \Rightarrow_S e$ is through property (3.16) which also requires opening and reestablishing the specification invariant to gain access to the global expression resource.

The logical relation. Now that we have seen how we can express relational properties as unary Hoare triples, we just need to integrate this idea with

the techniques from the previous section for translating region and effect annotations into specifications of abstract interference.

Consider the reference invariant, $\textsc{Ref}(r, \phi, x)$. In the unary setting it asserts ownership of a proxy for the underlying heap location that, depending on ownership of the per-location read and write tokens, contains a well-typed value and may be updated. In the binary setting, x is now a pair of locations (x_I, x_S) and the invariant asserts ownership of proxies for both the implementation and specification side heaps, but otherwise the structure of the definition remains the same. The binary reference invariant is defined in Figure 38. We use x_I and x_S as shorthand for the first and second projection of a pair x. Note that, in the binary setting, per-location read, write and no-read tokens are now indexed by a pair of locations, rather than just a single location. The $[\textsc{Rd}(x_I, x_S)]_r$ token now expresses that if locations x_I and x_S are related and belong to region r, then they contain related values, and likewise for the other tokens.

The LR_{BIN} logical relation satisfies the fundamental theorem of logical relations (Theorem 3.3.2), which expresses that all well-typed terms are related to themselves. It is also sound with respect to contextual approximation (Theorem 3.3.3).

Theorem 3.3.2 (Fundamental Theorem)**.** *If* $\Pi \mid \Delta \mid \Gamma \vdash e : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \models_{\text{BIN}} e \leq_{log} e : \tau, \varepsilon$

Theorem 3.3.3 (Soundness)**.** *If* $\Pi \mid \Delta \mid \Gamma \models_{\text{BIN}} e_I \leq_{log} e_S : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_I \leq_{ctx} e_S : \tau, \varepsilon$.

Binary Relation for $\lambda_{ref,conc}$ with Effects Using Multiple Simulations

The LR_{BIN} relation supports proofs of contextual approximations by showing that each step on the left can be simulated on the right. However, it requires that each thread on the left owns the local expression resource of the thread on the right that simulates the thread on the left. This is too restrictive in cases where multiple threads on the left are simulated by a single thread on the right, such as the case of parallelization. In this section we introduce our final logical relation, LR_{PAR}, that removes this restriction.

The idea is simple: The LR_{BIN} relation allowed us to reason about a single simulation; now, we generalize the relation to allow reasoning about multiple simulations, such that multiple threads on the left can be given ownership of an expression resource for the same thread on the right, in different simulations.

Multiple simulations. To make this precise, we generalize the existing specification ghost resources, so that we can have multiple independent copies, by

indexing the global and local expression resources ($mctx(e,\zeta)$ and $j \overset{\zeta}{\Rightarrow}_S e$) and heap resources ($heap_S(h,\zeta)$ and $l \mapsto^{\zeta}_S v$) with a simulation identifier ζ. For each simulation identifier ζ, the resources $mctx(e,\zeta)$ and $j \overset{\zeta}{\Rightarrow}_S e$ satisfy the same properties as before (properties (3.15) to (3.17)) and likewise for the heap resources (properties (3.18) and (3.19)). We can allocate new expression and heap resources initialized with an arbitrary expression e and an empty heap:

$$\top \Rrightarrow \exists \zeta.\, mctx(e,\zeta) * heap_S([],\zeta) * 0 \overset{\zeta}{\Rightarrow}_S e \tag{3.21}$$

The idea is to relate e_I and e_S if e_S can simulate any step performed by e_I in an arbitrary simulation ζ in which e_S appears in an evaluation context:

$$\forall \zeta, i, h_0, e_0.\, \boxed{\exists h, e.\, heap_S(h,\zeta) * mctx(e,\zeta) * (h_0; e_0 \rightarrow^* h; e)}^{\mathrm{SP}(\zeta)} \\ \vdash \{i \overset{\zeta}{\Rightarrow}_S e_S\}\, e_I\, \{v_I.\, \exists v_S.\, i \overset{\zeta}{\Rightarrow}_S v_S\}$$

This allows the caller of e_I to choose in which simulation ζ the specification e_S must simulate e_I. It also allows e_I to simulate sub-expressions of e_I in different simulations than ζ, provided it can still prove a simulation in ζ at the end.

We can allocate a new simulation with an arbitrary initial configuration $h;e$ and take ownership of the local heap and expression resources for this simulation, using (3.21) and (3.20).

This ability to simulate sub-expressions in different simulations is exactly what we need to disentangle an execution of $e_1 \| e_2$ into two independent executions of e_1 and e_2, when proving parallelization. To show that $e_1 \| e_2$ is related to (e_1, e_2) in the expression relation, we (roughly) prove the following triple:

$$\boxed{\exists h, e.\, heap_S(h,\zeta) * mctx(e,\zeta) * (h_0; e_0 \rightarrow^* h; e)}^{\mathrm{SP}(\zeta)} \vdash \\ \{i \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * \cdots\}\, e_1 \| e_2\, \{v_I.\, \exists v_S.\, i \overset{\zeta}{\Rightarrow}_S v_S * \cdots\}$$

Recall from the Introduction that the idea is to use the effect annotations to prove that an execution of $e_1 \| e_2$ can be disentangled into semi-independent executions of e_1 and e_2.

Since e_1 and e_2 are well-typed, it follows by the fundamental theorem of logical relations that they are related to themselves. To use these assumptions we must pass ownership of a local expression resource to each of e_1 and e_2 with e_1 and e_2 in an evaluation context, respectively. We could use the ζ simulation with e_1 since e_1 is already in an evaluation context in the ζ simulation. However, this leaves us without an expression resource for e_2.

Instead, the idea is to suspend the ζ simulation and create two new simulations ζ_1 and ζ_2 with e_1 as the full specification of the ζ_1 simulation and e_2 as

the full specification of the ζ_2 simulation. Then we can appeal to relatedness of e_1 and e_2 to themselves with ζ_1 and ζ_2 as the respective simulations, which will show the existence of the two independent executions of e_1 and e_2. Once $e_1 \parallel e_2$ has terminated on the left, we can resume the ζ simulation and use the two independent executions of e_1 and e_2 to take the appropriate steps in the ζ simulation.

The reason this works, is the effect annotations, which ensure that e_1 and e_2 are semi-independent. In particular, all locations accessed by both e_1 and e_2 are read-only and can therefore soundly be shared between the ζ_1 and ζ_2 simulations.

Relating heaps in multiple simulations. In previous relations, the region invariant ensured that all specification heap proxies ($l \overset{\pi}{\hookrightarrow}_{S,r} v$) matched the contents of the actual specification heap. With multiple simulations, we have multiple specification heaps. The idea is to allow proxies to be tied to multiple specification heaps, provided we can guarantee that the given references are immutable. In cases where we cannot guarantee immutability, we still only allow proxies to be tied to a single simulation, to ensure we can reason locally about reductions in simulations.

To capture this formally, we introduce a new ghost resource, to specify whether a region is immutable or not, and which simulations the region proxies are tied to. The $[\textsc{Im}(r, S, h)]^\pi$ resource asserts that region r is immutable and the current specification heap of the region is h, while $[\textsc{Mu}(r, S)]^\pi$ asserts that it is mutable. In both cases the set S specifies which simulations the proxies of region r are tied to. In the mutable state, we require that the set S is a singleton. We call this ghost resource the *specification link* resource.

The fractional permission is used to track whether we are allowed to change the state of a region. If we own a specification link resource exclusively (i.e., $\pi = 1$), then we can change its state between mutable and immutable and which simulations the proxies of the region are tied to.

$$[\textsc{Mu}(r, S)]^1 \Longleftrightarrow [\textsc{Im}(r, S', h)]^1 \tag{3.22}$$

Both tokens can be split arbitrarily using the fraction. Any two fractional immutable tokens must agree on the current heap and which simulations the region is tied to:

$$\begin{aligned}&[\textsc{Im}(r, S_1, h_1)]^{\pi_1} * [\textsc{Im}(r, S_2, h_2)]^{\pi_2} \implies \\ &\quad [\textsc{Im}(r, S_1, h_1)]^{\pi_1} * [\textsc{Im}(r, S_2, h_2)]^{\pi_2} * (h_1 = h_2) * (S_1 = S_2)\end{aligned} \tag{3.23}$$

Disjointness of allocations. We need two final bits of ghost state before we can define the full logical relation. Namely, we need a way to control which locations simulations use when allocating new locations.

To facilitate this level of control over locations, we introduce a ghost resource, $[X]$, for asserting ownership of a set of locations X. These can be split and recombined and ensure that disjoint resources refer to disjoint sets of locations:

$$[X_1 \uplus X_2] \Longleftrightarrow [X_1] * [X_2] \tag{3.24}$$

$$[X_1] * [X_2] \implies [X_1] * [X_2] * X_1 \cap X_2 = \emptyset \tag{3.25}$$

The idea is to give each specification invariant ownership of a countably infinite set of locations that only that simulation may use for future allocations.

To allow simulations to replay reductions from other simulations, we also need a way of deactivating a simulation, such that we can take back ownership of that simulation's locations. To achieve this we introduce a ghost resource $[\textsc{Sr}]^{\pi}_{\zeta}$, which we refer to as a *specification runner resource*, to track whether a simulation is active. Ownership of any fraction of this token witnesses that the simulation is active.

The LR_{PAR} logical relation is defined in Figure 39. The most important difference compared to the LR_{BIN} relation, is in the $locs(r)$ predicate contained in the region invariant Reg. Reg now asserts fractional ownership of a specification link resource for the given region through the *slink* predicate. In case the region is immutable, the pair of heaps given by the specification link resource must match the actual implementation and specification heap for the references belonging to the given region. The region invariant further asserts ownership of the local specification heap resource $l \mapsto^{\zeta}_{S} v$ for every simulation $\zeta \in S$ tied to the given region through the specification link resource.

The specification invariant, Spec, has been extended to support global freshness when allocating, as explained above. Either the specification invariant owns half of the specification runner resource, in which case it also asserts ownership of countably infinite sets of fresh locations through the $disj$ predicate. Otherwise, the specification invariant is inactive and asserts exclusive ownership of the specification runner resource.

Finally, the expression relation now asserts fractional ownership of the specification runner resource and fractional ownership of specification link resources, for all regions in the context. The specification runner resource ensures that the ζ simulation is active. In case a region is private or the effect mask contains a write or allocation effect for the given region, then the region must be in the mutable state and tied only to the simulation ζ. Otherwise, the region may be in either the mutable or the immutable state, as long as it is tied to the ζ simulation.

The LR_{PAR} relation is sound with respect to contextual approximation and supports parallelization.

Theorem 3.3.4 (Soundness). *If* $\Pi \mid \Delta \mid \Gamma \vDash_{\text{PAR}} e_I \leq_{log} e_S : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_I \leq_{ctx} e_S : \tau, \varepsilon$.

New predicates

$$P_{par}(R,g,\varepsilon,M,\zeta) \triangleq \circledast_{\rho\in mutable(R,g,\varepsilon)}[\textsc{Mu}(M(\rho),\{\zeta\})]^{g(\rho)} *$$
$$\circledast_{\rho\in R\setminus mutable(R,g,\varepsilon)}\exists h,S.slink(M(\rho),\{\zeta\}\uplus S,h,g(\rho),g(\rho))$$
$$slink(r,S,h,\pi,\pi') \triangleq [\textsc{Mu}(r,S)]^{\pi} \vee [\textsc{Im}(r,S,h)]^{\pi'}$$
$$disj(X_0,X) \triangleq \exists Y.\ [Y] \wedge \mathrm{dom}(X_0) \cap Y = \emptyset \wedge (\mathrm{dom}(X) \setminus \mathrm{dom}(X_0)) \subset Y$$
$$mutable(R,g,\varepsilon) \triangleq \mathsf{wrs}\ \varepsilon \cup \mathsf{als}\ \varepsilon \cup \left\{\rho \mid \rho \in R \wedge g(\rho) = \tfrac{1}{2}\right\}$$

Changes to previous definitions

$$locs(r) \triangleq \exists h,S.\ slink(r,S,h_S,\tfrac{1}{2},\tfrac{1}{4}) * rheap_I(h_I,r) * rheap_S(h_S,r) *$$
$$alloc(h,r) * \circledast_{(l,v)\in h_I} l \mapsto_I v * \circledast_{\zeta\in S}\ \circledast_{(l,v)\in h_S} l \mapsto_S^{\zeta} v$$
$$\textsc{Spec}(h_0,e_0,\zeta) \triangleq \exists h,e,\pi.\ heap_S(h,\zeta) * mctx(e,\zeta) * (h_0,e_0) \to^* (h,e) *$$
$$([\textsc{Sr}]_{\zeta}^{1} \vee ([\textsc{Sr}]_{\zeta}^{\frac{1}{2}} * disj(h_0,h_S)))$$
$$P_{reg}(\cdots,\zeta) \triangleq \cdots * P_{par}(R,\tfrac{\mathbf{1}}{2}\circ g,\varepsilon,M,\zeta)$$
$$\mathcal{E}_{\varepsilon,M}^{\Pi,\Lambda}(\phi)(e_I,e_S) \triangleq \forall g,j,h_0,e_0,\pi,\zeta.\ \boxed{\textsc{Spec}(h_0,e_0,\zeta)} \vdash$$
$$\left\{j \overset{\zeta}{\Rightarrow}_S e_S * [\textsc{Sr}]_{\zeta}^{\pi} * P_{reg}(\Lambda,\mathbf{1},\varepsilon,M,\zeta) * P_{reg}(\Pi,g,\varepsilon,M,\zeta)\right\}$$
$$e_I$$
$$\left\{\begin{array}{l} v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]_{\zeta}^{\pi} * \phi(v_I,v_S) * \\ P_{reg}(\Lambda,\mathbf{1},\varepsilon,M,\zeta) * P_{reg}(\Pi,g,\varepsilon,M,\zeta) \end{array}\right\}_{\top}$$

Figure 39: $\mathrm{LR}_{\mathrm{PAR}}$: Binary rel. for $\lambda_{ref,conc}$ with effect-types and effect-based simulations.

Theorem 3.3.5 (Parallelization (semantically))**.** *If*

1. $\mathcal{E}_{\varepsilon_1,M}^{\Lambda_3,\Lambda_1}([\![\tau_1]\!]^M)(e_{1I},e_{1S})$ *and* $\mathcal{E}_{\varepsilon_2,M}^{\Lambda_3,\Lambda_2}([\![\tau_2]\!]^M)(e_{2I},e_{2S})$
2. $\mathsf{als}\ \varepsilon_1 \cup \mathsf{wrs}\ \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{als}\ \varepsilon_2 \cup \mathsf{wrs}\ \varepsilon_2 \subseteq \Lambda_2$
3. $\mathsf{rds}\ \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ *and* $\mathsf{rds}\ \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then $\mathcal{E}_{\varepsilon_1\cup\varepsilon_2,M}^{-,(\Lambda_1,\Lambda_2,\Lambda_3)}([\![\tau_1\times\tau_2]\!]^M)(e_{1I}\,\|\,e_{2I},(e_{1S},e_{2S}))$.

To illustrate how we can use the $\mathrm{LR}_{\mathrm{PAR}}$ relation to prove contextual refinements that depend on the effect annotations, we give a proof sketch of Theorem 3.1.3 below. The full proof can be found in §II.

Recall that Theorem 3.1.3 states that each of the two stack modules is contextually equivalent to their counterpart without a CAS loop, at an effect

$$
\begin{aligned}
stack_{nc_1}() &= \mathsf{let}\ h = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ (push_{nc_1}(h), pop_{nc_1}(h)) \\
push_{nc_1}(h) &= \mathsf{rec}\ p(n).h := \mathsf{inj}_2\ (n, !h) \\
pop_{nc_1}(h) &= \mathsf{rec}\ p(\_).\mathsf{case}(!h, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\
&\qquad\qquad \mathsf{inj}_2\ (n, v') \Rightarrow h := v'; \mathsf{inj}_2\ n)
\end{aligned}
$$

Figure 310: Stack module without CAS.

type where the local state of the stack module belongs to a private region:

$$
\begin{aligned}
&- \mid \rho \mid - \vdash stack\_nc_1 \cong_{ctx} stack_1 : \tau'_{\textsc{Stack}}, \left\{al_\rho\right\}\ \text{and} \\
&- \mid \rho \mid - \vdash stack\_nc_2 \cong_{ctx} stack_2 : \tau'_{\textsc{Stack}}, \left\{al_\rho\right\}\ \text{with} \\
&\tau'_{\textsc{Stack}} = \mathbf{1} \to^{-,\rho}_{\{al_\rho\}} (\mathsf{int} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})
\end{aligned}
$$

We will focus on the first contextual equivalence where the $stack_{nc_1}$ module is implemented using a single reference to a pure functional list as shown in Figure 310.

To show logical relatedness between $stack_{nc_1}$ and $stack_1$ we will have to assert a relation between the state maintained by the modules. Since the state is local to each module we are not required to use the $[\![\mathsf{ref}_\rho\ \tau]\!]^M$ interpretation and are free to pick any invariant to relate the state of the two modules.

A suitable relation would assert ownership of each head-pointer for the region ρ and would state that each pair-wise entry in the stacks are related. Stack is a promising candidate:

$$
\begin{aligned}
\textsc{Stack}(h, r, l, v) &\triangleq h_I \overset{1}{\hookrightarrow}_{I,r} v_I * h_S \overset{1}{\hookrightarrow}_{S,r} v_S * vals(l, v) \\
vals(\mathsf{nil}, v) &\triangleq v_I = \mathsf{inj}_1\ () \wedge v_S = \mathsf{inj}_1\ () \\
vals(x :: xs, v) &\triangleq [\![\mathsf{int}]\!](x) * \exists v'.\ v_I = \mathsf{inj}_2\ (x_I, v'_I) \wedge \\
&\qquad v_S = \mathsf{inj}_2\ (x_S, v'_S) \wedge vals(xs, v')
\end{aligned}
$$

Note that while we are free to pick an invariant to relate the internal state of the modules, we still use the points-to proxy resources to ensure that the state is tied to simulations correctly.

The Stack relation allows us to read from the head-pointer using a simple extension of property 3.12:

$$
\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash \{\textsc{Stack}(h, r, l, v)\}\ !h_I\ \{v_I. \exists v_S. \textsc{Stack}(h, r, l, (v_I, v_S))\} \tag{3.26}
$$

Similarly, we can use a variant of property 3.13 to do assignment to the head-pointer location:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash \{\textsc{Stack}(h,r,l,v)\} \quad h_I := v' \quad \{h_I \overset{1}{\hookrightarrow}_{I,r} v' * h_S \overset{1}{\hookrightarrow}_{S,r} v_S * vals(l,v)\} \tag{3.27}$$

Putting Stack into an invariant is not sufficient, however, for showing the direction $stack_{nc_1} \leq_{log} stack_1$. The reason is that both $push_1(h_S)(n_S)$ and $pop_1(h_S)()$ has a **CAS** operation, that we must guarantee succeeds in a compatible state with $push_{nc_1}(h_I)(n_I)$ and $pop_{nc_1}(h_I)()$, for related n_I and n_S. This relies on the fact that the ρ region is private which ensures that the environment cannot access the local state during the stack operations. We can capture this by defining a Rel predicate with a property that allow us to exchange the exclusive write permission $[\textsc{Wr}]^1_r$ for ownership of the stack module's points-to proxies:

$$\boxed{\textsc{Rel}(h,r)} \vdash [\textsc{Wr}]^1_r \Longleftrightarrow \exists l, v.\, \textsc{Stack}(h,r,l,v) \tag{3.28}$$

We will need to establish the invariant when the data structures in the implementation and specification side are both empty:

$$\vdash h_I \overset{1}{\hookrightarrow}_{I,r} \mathsf{inj}_1\ () * h_S \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_1\ () \Rrightarrow \boxed{\textsc{Rel}(h,r)} \tag{3.29}$$

Rel as defined below allows for the above view-shifts:

$$\textsc{Rel}(h,r) \triangleq \exists l, v.\, \textsc{Stack}(h,r,l,v) \vee [\textsc{Wr}]^1_r$$

For this particular example we have no need for interpreting read effects on the local state. The Rel invariant therefore makes no mention of the $[\textsc{Rd}]^1_r$ token.

We show logical equivalence by showing logical approximation in both directions. Here we present a proof outline of the direction $stack_{nc_1} \leq_{ctx} stack_1$, the full proof of both directions can be found in §II. Since $stack_{nc_1}$ and $stack_1$ are already values, it suffices to show they are related in the value relation for $\tau'_{\textsc{Stack}}$, which reduces to showing that:

$$\mathcal{E}^{-,\rho}_{\{al_\rho\},M}(\phi)(stack_{nc_1}(), stack_1())$$

where $\phi = [\![(\mathsf{int} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})]\!]^M$.

Thus we first show that we can establish the Rel invariant using the local state allocated by the two modules. Next, we show that this invariant is preserved by the push and pop operations and that they are pairwise related assuming this invariant. The proof outline is given below and uses the

following two properties to allocate points-to proxies on the implementation and specification side, respectively:

$$\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash \{[\textsc{Al}]^{\pi}_{r}\}\ \mathsf{new}\ v\ \{y.\ [\textsc{Al}]^{\pi}_{r} * h_I \overset{1}{\hookrightarrow}_{I,r} v\} \tag{3.30}$$

$$\begin{aligned}\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash\ & [\textsc{Al}]^{\pi}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\pi} * i \overset{\zeta}{\Rightarrow}_{S} \mathsf{new}\ v \\ \Rrightarrow\ & \exists l.\ [\textsc{Al}]^{\pi}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\pi} * i \overset{\zeta}{\Rightarrow}_{S} l * l \overset{1}{\hookrightarrow}_{S,r} v\end{aligned} \tag{3.31}$$

Context: $\boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, [\![\mathbf{1}]\!]^{M}(y)$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_{S} \mathsf{let}\ h_S = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ (push_1(h_S), pop_1(h_S)) * \\ [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Al}]^{1}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} \end{array}\right\}$$

$$\left\{\begin{array}{l} \exists i.\ j \overset{\zeta}{\Rightarrow}_{S} \mathsf{let}\ h_S = i\ \mathsf{in}\ (push_1(h_S), pop_1(h_S)) * \\ i \overset{\zeta}{\Rightarrow}_{S} \mathsf{new}\ \mathsf{inj}_1\ () * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Al}]^{1}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} \end{array}\right\} \quad \mathsf{new}\ \mathsf{inj}_1\ ()$$

// Follows from Properties 3.30 and 3.31

$$\left\{\begin{array}{l} h_I.\ \exists h'_S, i.\ j \overset{\zeta}{\Rightarrow}_{S} \mathsf{let}\ h_S = i\ \mathsf{in}\ (push_1(h_S), pop_1(h_S)) * i \overset{\zeta}{\Rightarrow}_{S} h'_S * \\ [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Al}]^{1}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} * h_I \overset{1}{\hookrightarrow}_{I,r} \mathsf{inj}_1\ () * h_S \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_1\ () \end{array}\right\}$$

// Follows from Property 3.29

$$\left\{\begin{array}{l} \exists h_I, h'_S.\ j \overset{\zeta}{\Rightarrow}_{S} \mathsf{let}\ h_S = h'_S\ \mathsf{in}\ (push_1(h_S), pop_1(h_S)) * \\ [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Al}]^{1}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} * \boxed{\textsc{Rel}((h_I,h'_S),r)} \end{array}\right\}$$

$$\left\{\begin{array}{l} \exists h.\ j \overset{\zeta}{\Rightarrow}_{S} (push_1(h_S), pop_1(h_S)) * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Al}]^{1}_{r} * \\ [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} * \boxed{\textsc{Rel}(h,r)} \end{array}\right\}$$

$$(push_{nc_1}(h_I), pop_{nc_1}(h_I))$$

$$\left\{\begin{array}{l} v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_{S} v_S * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Al}]^{1}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} * \\ [\![(\mathsf{int} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1}) \times (\mathbf{1} \to^{-,\rho}_{\{wr_\rho, rd_\rho\}} \mathbf{1} + \mathsf{int})]\!]^{M}(v_I, v_S) \end{array}\right\}$$

For the last step we need to show the following two refinements:

$$\boxed{\textsc{Rel}(h,r)}, [\![\mathsf{int}]\!]^{M}(n) \vdash \mathcal{E}^{-,\rho}_{\{wr_\rho, rd_\rho\},M}([\![\mathbf{1}]\!]^{M})(push_{nc_1}(h_I)(n_I), push_1(h_S)(n_S))$$

$$\boxed{\textsc{Rel}(h,r)} \vdash \mathcal{E}^{-,\rho}_{\{wr_\rho, rd_\rho\},M}([\![1 + \mathsf{int}]\!]^{M})(pop_{nc_1}(h_I)(), pop_1(h_S)())$$

We sketch a proof of the first refinement below. The proof of the second refinement can be found in §II.

The interpretation of the region and effect annotations for the first refinement is as follows. Since we do not interpret read effects on the local state, $[\textsc{Rd}]^{1}_{r}$ is framed off in the proof outline below.

$$\begin{aligned} & P_{reg}(\{\rho\}, \mathbf{1}, \{wr_\rho, rd_\rho\}, M[\rho \mapsto r], \zeta) \\ & \quad = [\textsc{Wr}]^{1}_{r} * [\textsc{Rd}]^{1}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{1}{2}} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \end{aligned}$$

The proof starts by trading the exclusive write token for ownership of the local state (3.28). The we use (3.27) to push n_I onto the implementation-side stack.

$$\text{Context: } \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Rel}(h, r)}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, [\![\mathsf{int}]\!](n)$$
$$\left\{ j \overset{\zeta}{\Rightarrow}_S push_1(h_S)(n_S) * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Wr}]^{1}_{r} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} \right\}$$
$$\left\{ \exists l, v.\ j \overset{\zeta}{\Rightarrow}_S push_1(h_S)(n_S) * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * \textsc{Stack}(h, r, l, v) \right\}$$
$$!h_I$$
$$\left\{ v_I. \exists l, v_S.\ j \overset{\zeta}{\Rightarrow}_S push_1(h_S)(n_S) * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * \textsc{Stack}(h, r, l, (v_I, v_S)) \right\}$$
$$h_I := \mathsf{inj}_2\ (n_I, v_I)$$
$$\left\{ \begin{array}{l} v_I'. \exists l, v_S.\ j \overset{\zeta}{\Rightarrow}_S push_1(h_S)(n_S) * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * \\ h_I \overset{1}{\hookrightarrow}_{I,r} \mathsf{inj}_2\ (n_I, v_I) * h_S \overset{1}{\hookrightarrow}_{S,r} v_S * vals(l, v) * v_I' = () \end{array} \right\}$$

After pushing n_I on the stack on the implementation side, we simulate pushing n_S on the specification side. Let:

$$\begin{aligned} K_1 &\triangleq \mathsf{let}\ v = [\,]\ \mathsf{in} \\ &\qquad \mathsf{if}\ \mathsf{CAS}(h_S, v, \mathsf{inj}_2\ (n_S, v))\ \mathsf{then}\ ()\ \mathsf{else}\ loop(n_S) \\ K_2 &\triangleq \mathsf{if}\ [\,]\ \mathsf{then}\ ()\ \mathsf{else}\ loop(n_S) \end{aligned}$$

be the evaluation contexts which require a non-trivial reduction. Notice that $push_1(h_S)(n_S) = K_1[!h_S]$. We can now perform the simulation:

$$\begin{aligned} &\boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash \\ &\quad j \overset{\zeta}{\Rightarrow}_S push_1(h_S)(n_S) * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * h_S \overset{1}{\hookrightarrow}_{S,r} v_S \\ &\Rrightarrow \exists i.\ j \overset{\zeta}{\Rightarrow}_S K_1[i] * i \overset{\zeta}{\Rightarrow}_S !h_S * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * h_S \overset{1}{\hookrightarrow}_{S,r} v_S \\ &\Rrightarrow \exists i.\ j \overset{\zeta}{\Rightarrow}_S K_2[i] * i \overset{\zeta}{\Rightarrow}_S \mathsf{CAS}(h_S, v_S, \mathsf{inj}_2\ (n_S, v_S)) * [\textsc{Sr}]^{\pi}_{\zeta} * \\ &\quad [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * h_S \overset{1}{\hookrightarrow}_{S,r} v_S \\ &\Rrightarrow j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * h_S \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_2\ (n_S, v_S) \end{aligned}$$

The simulation follows from repeatedly stepping by using property 3.15 and 3.16. Observe that $\mathsf{CAS}(h_S, v_S, \mathsf{inj}_2\ (n_S, v_S))$ always succeeds since we have ownership of $h_S \overset{1}{\hookrightarrow}_{S,r} v_S$. We can now finish the proof by reestablishing the relation between the local state of the modules and trading it for the exclusive

write permission:

$$\left\{\begin{array}{l}\exists l, v_S.\ j \overset{\zeta}{\Rightarrow}_S\ () * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * \\ h_I \overset{1}{\hookrightarrow}_{I,r} \mathsf{inj}_2\ (n_I, v_I) * h_S \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_2\ (n_S, v_S) * vals(l, v)\end{array}\right\}$$

$$\left\{\begin{array}{l}\exists l, v'.\ j \overset{\zeta}{\Rightarrow}_S\ () * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * h_I \overset{1}{\hookrightarrow}_{I,r} v'_I * h_S \overset{1}{\hookrightarrow}_{S,r} v'_S * \\ vals(n :: l, v')\end{array}\right\}$$

$$\left\{\exists l, v'.\ j \overset{\zeta}{\Rightarrow}_S\ () * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}} * \textsc{Stack}(h, r, n :: l, v')\right\}$$

$$\left\{\exists l, v'.\ j \overset{\zeta}{\Rightarrow}_S\ () * [\textsc{Sr}]^{\pi}_{\zeta} * [\textsc{Wr}]^{1}_{r} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}}\right\}$$

3.4 Discussion

We have already mentioned some related work along the way; here we discuss some other related work.

Benton *et al.* initiated a line of work on relational models of type-and-effect systems to formally justify effect-based program transformations for increasingly sophisticated sequential programming languages and increasingly expressive effect systems [8, 10–13, 74]. Birkedal *et al.* showed how to extend this approach to a concurrent language [14]. The effect system we use here is from *loc. cit.* Birkedal *et al.*'s relational interpretation is defined by a concrete step-indexed Kripke logical relation. They used the model to prove a parallelization theorem similar to ours, but the proof was very technical and consisted of manual disentangling and re-ordering of computation steps. Part of the reason for this was that support for parallelization was not built into their logical relation and had to be proven separately. In contrast, we build in support for parallelization in the $\mathrm{LR}_{\textsc{Par}}$ relation through its support for multiple simulations. This allows us to reduce the proof of the parallelization theorem to the essence of why it holds: framing. Moreover, as mentioned in the Introduction, it makes it possible to use the program logic to show that an expression satisfies the semantic invariants imposed by the type system even if the expression is not statically well-typed and to reason about refinements.

In recent work, Benton *et al.* [9] have also considered a concurrent language, which in contrast to the language considered here only includes first-order store. Technically, this makes the construction of a logical relations model simpler, since one avoids having to deal with the type-world circularity mentioned in the Introduction. Their type-and-effect system does not support dynamic allocation of abstract locations (which correspond to regions in our setup), requiring all abstract locations to be given up front. Our type-and-effect system supports dynamic allocation and hiding of regions, through the masking rule. On the other hand, their effect system supports a notion of abstract effects, which means, *e.g.*, that an operation in a data

structure module can be considered pure even if it uses effects internally, as long as those effects are not observable outside the module boundary. Benton *et al.* use this facility for treating refinement of fine-grained concurrent data structures, illustrated using an idealized Michael-Scott queue. Our semantics also supports refinements between fine-grained concurrent data structures, using Iris' support for general invariants. In this paper we have focused on an example of a refinement that only holds by restricting interference through the type-and-effect system. Our method also scales to fine-grained concurrent data structures that use helping, thanks to Iris [33].

Raza *et al.* [64] and Botincan *et al.* [16], both explore automatic parallelization of sequential programs verified in separation logic. Raza *et al.* rely on specifications inferred from a shape analysis. Botincan *et al.* explore the idea of using the proof to insert synchronization that ensures the dependencies of the original program are preserved. These analyses focus on first-order programs, whereas our type-and-effect system applies to higher-order programs.

The idea of defining logical relations in a program logic goes back at least to Plotkin and Abadi, who used a second-order logic to define logical relations for a second-order lambda calculus [58]. Dreyer *et al.* used a second-order logic with a Löb modality, inspired by [7], to give a logical relations interpretation of a programming language with recursive types [21]. The logic used by Dreyer *et al.* did not support invariants and hence it did not support the interpretation of reference types. Turon *et al.* showed how to use a variant of second-order concurrent separation logic with invariants for giving a logical relations interpretation of an ML-like type system for a language similar to the one considered in this paper [78]. To define logical relations in the unary separation logic, their logic had a *built-in* notion of specification resources and a single specification invariant. In contrast, here we use a higher-order concurrent separation logic, Iris, which is flexible enough that one can define specification resources and invariants in it. We rely crucially on this flexibility for the $\mathrm{LR}_{\mathrm{PAR}}$ relation to support multiple simulations, as discussed in Section 3.3.

Chapter 4

A Logical Relation for Monadic Encapsulation of State

Proving contextual equivalences in the presence of runST

AMIN TIMANY, imec-Distrinet, KU Leuven, Belgium
LÉO STEFANESCO, IRIF, Université Paris Diderot & CNRS, France
MORTEN KROGH-JESPERSEN, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

In Proceedings of the ACM on Programming Languages (POPL), 2018.

The formal development accompanying this research project can be found on the Iris project web-site https://iris-project.org. As of writing a direct link for the development is https://iris-project.org/artifacts/2018-popl-runst.tar.gz.

Abstract

We present a logical relations model of a higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with `runST`. We use our logical relations model to show that `runST` provides proper encapsulation of state, by showing that effectful computations encapsulated by `runST` are heap independent. Furthermore, we show that contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of `runST`. This is the first time such relational results have been proven for a language with monadic encapsulation of state. We have formalized all the technical development and results in Coq.

4.1 Introduction

Haskell is often considered a *pure* functional programming language because effectful computations are encapsulated using monads. To preserve purity, values usually cannot escape from those monads. One notable exception is the ST monad, introduced by Launchbury and Jones [47]. The ST monad comes equipped with a function $\mathtt{runST} : (\forall \beta, \mathtt{ST}\ \beta\ \tau) \to \tau$ that allows a value to escape from the monad: `runST` runs a stateful computation of the monadic type $\mathtt{ST}\ \beta\ \tau$ and then returns the resulting value of type τ. In the original paper [47], the authors argued informally that the ST monad is "safe", in the sense that stateful computations are properly encapsulated and therefore the purity of the functional language is preserved.

In this paper we present a logical relations model of STLang, a call-by-value higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with `runST`. In contrast to earlier work, the operational semantics of STLang uses a *single global mutable heap*, capturing how the language would be implemented in reality. We use our logical relations model to show, for the first time, that `runST` provides proper encapsulation of state. Concretely, we state a number of contextual refinements and equivalences that are expected to hold for pure computations and we then use our logical relations model to prove that they indeed hold for STLang, i.e., in the presence of stateful computations encapsulated using `runST`. Moreover, we show a State-Independence theorem that intuitively expresses that, for any well-typed expression e of type τ, the evaluation of e in a heap h is independent of the choice of h, i.e., e cannot read from or write to locations in h but may allocate new locations (via encapsulated stateful computations). Note that this is the strong result one really wishes to have since it is proved for a standard operational semantics using a single global mutable heap allowing for updates in-place, not an abstract semantics partitioning memory into disjoint regions as some earlier work [48, 52].

In STLang, values of any type can be stored in the heap, and thus it is an example of a language with so-called higher-order store. It is well-known that it is challenging to construct logical relations for languages with higher-order store. We define our logical relations model in Iris, a state-of-the-art higher-order separation logic [33, 34, 38]. Iris's base logic [38] comes equipped with certain modalities which we use to simplify the construction of the logical relation. Logical relations for other type systems have recently been defined in Iris [39, 42], but to make our logical relations model powerful enough to prove the contextual equivalences for purity, we use a new approach to defining logical relations in Iris, which involves several new technical innovations, described in §4.3 and §4.5.

Another reason for using Iris is that the newly developed powerful proof mode for Iris [39] makes it possible to conduct interactive proofs in the Iris logic in Coq, much in the same way as one normally reasons in the Coq logic

itself. Indeed, we have used the Iris proof mode to formalize all the technical results in this paper in Iris in Coq.

In the remainder of this Introduction, we briefly recap the Haskell ST monad and recall why runST intuitively encapsulates state. We emphasize that STLang, unlike Haskell, is call-by-value; we show Haskell code to make the examples easier to understand. Finally, we give an overview of the technical development and our new results.

A Recap of the Haskell ST Monad

The ST monad, as described in [47] and implemented in the standard Haskell library, is actually a family ST β of monads, where β ranges over types, which satisfy the following interface. The first two functions

```
return :: α → ST β α
(>>=)  :: ST β α → (α → ST β α') → ST β α'
```

are the standard Kleisli arrow interface of monads in Haskell; »= is pronounced "bind". Recall that in Haskell, free type variables (α, α', and β above) are implicitly universally quantified.[1]

The next three functions

```
newSTRef   :: α → ST β (STRef β α)
readSTRef  :: STRef β α → ST β α
writeSTRef :: STRef β α → α → ST β ()
```

are used to *create*, *read from* and *write into* references, respectively. Notice that the reference type STRef β τ, contains the type of the contents of the reference cells, τ, but also another type parameter, β, which, intuitively, indicates which (logical) region of the heap this reference belongs to. The interesting part of the interface is the interaction of this type parameter with the following function

```
runST :: (∀ β. ST β α) → α
```

The runST function runs effectful computations and extracts the result from the ST monad. Notice the impredicative quantification of the type variable of runST.

Finally, equality on references is decidable:

```
(==) :: STRef β α → STRef β α → bool
```

Notice that equality is an ordinary function, since it returns a boolean value directly, not a value of type ST β bool.

[1] In STLang, we use capital letters, e.g. X, for type variables and use ρ for the index type in ST ρ τ and STRef ρ τ.

```
fibST :: Integer → Integer
fibST n =
  let fibST' 0 x _ = readSTRef x
      fibST' n x y = do
          x' <- readSTRef x
          y' <- readSTRef y
          writeSTRef x y'
          writeSTRef y (x'+y')
          fibST' (n-1) x y
  in
  if n < 2 then n else
    runST  do
      x <- newSTRef 0
      y <- newSTRef 1
      fibST' n x y
```

```
let fibST : ℤ -> ℤ =
  let rec fibST' n x y =
    if n = 0 then !x
    else bind !x in λ x' ->
         bind !y in λ y' ->
         bind x := y' in λ () ->
         bind y := (x'+y') in λ () ->
            fibST' (n-1) x y
  in
  if n < 2 then n else
    runST {
      bind (ref 0) in λ x ->
      bind (ref 1) in λ y ->
        fibST' n x y }
```

Figure 41: Computing Fibonacci numbers using the ST monad in Haskell (left) and in STLang (right). Haskell code adapted from https://wiki.haskell.org/Monad/ST. do is syntactic sugar for wrapping bind around a sequence of expressions.

Figure 41 shows how to compute the n-th term of the Fibonacci sequence in Haskell using the ST monad and, for comparison, in our model language STLang. Haskell programmers will notice that the STLang program on the right is essentially the same as the one on the left after the do-notation has been expanded. The inner function `fibST'` can be typed as follows:

```
fibST' :: Integer → STRef β Integer → STRef β Integer → ST β Integer
```

Hence, the argument of `runST` has type `(∀ β. STRef β Integer)` and thus `fibST` indeed has return type `Integer`.

Encapsulation of State Using `runST`: What is the Challenge?

The operational semantics of the `newSTRef`, `readSTRef`, `writeSTRef` operations is intended to be the same as for ML-style references. In particular, an implementation should be able to use a global heap and in-place update for the stateful operations. The ingenious idea of Launchbury and Jones [47] is that the parametric polymorphism in the type for `runST` should still ensure that stateful computations are properly encapsulated and thus, that ordinary functions remain pure.

The intuition behind this intended property is that the first type variable parameter of ST, denoted β above, actually denotes a region of the heap, and that we can *imagine* that the heap consists of a collection of disjoint regions, named by types. A computation e of type ST β τ can then read, write, and allocate in the region named β, and then produce a value of type τ.

Moreover, if e has type $\forall\beta.\ \mathrm{ST}\ \beta\ \tau$, with β not free in τ, the intuition is that `runST` e can allocate a fresh region, which e may use and then, since β is not free in τ, the resulting value of type τ cannot involve references in the region β. It is therefore safe to discard the region β and return the value of type τ. Since stateful computations intuitively are encapsulated in this way, this should also entail that the rest of the "pure" language indeed remains pure. For example, it should still be the case that for an expression e of type τ, running e twice should be the same as running it once. More precisely, we would expect the following contextual equivalence to hold for any expression e of type τ:

$$\texttt{let}\, x = e\, \texttt{in}\, (x, x) \quad \approx_{\mathsf{ctx}} \quad (e, e) \tag{$*$}$$

Note that, of course, this contextual equivalence would not hold in the presence of unrestricted side effects as in ML: if `e` is the expression `y := !y + 1`, which increments the reference `y`, then the reference would be incremented by 1 on the left hand-side of ($*$) and by 2 on the right.

Similar kinds of contextual equivalences and refinements that we expect should hold for a pure language should also continue to hold. Moreover, we also expect that the State-Independence theorem described above should hold.

Notice that this intuitive explanation is just a conceptual model — *the real implementation of the language uses a standard global heap with in-place update and the* challenge *is to prove that the type system still enforces this intended proper encapsulation of effects*.

In this paper, we provide a solution to this challenge: we define a higher-order functional programming language, called STLang, with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with `runST`. The operational semantics uses a global mutable heap for stateful operations. We develop a logical relations model which we use to prove contextual refinements and equivalences that one expects should hold for a pure language in the presence of stateful computations encapsulated using `runST`.

Earlier work has focused on *simpler* variations of this challenge; specifically, it has focused on type safety, and none of the earlier formal models can be used to show expected contextual equivalences for the pure part of the language relative to an operational semantics with a single global mutable heap. In particular, the semantics and parametricity results of Launchbury and Peyton Jones [48] is denotational and does not use a global mutable heap with in-place update, and they state [48, Section 9.1] that proving that the remaining part of the language remains pure for an implementation with in-place update "would necessarily involve some operational semantics." We discuss other related work in §4.7.

Overview of Results and the Technical Development

In §4.2 we present the operational semantics and the type system for our language STLang. In this paper, we focus on the encapsulation properties of a Haskell-style monadic type system for stateful computations. The choice of evaluation order is an orthogonal issue and, for simplicity (to avoid having to formalize a lazy operational semantics), we use call-by-value left-to-right evaluation order. Typing judgments take the standard form $\Xi \mid \Gamma \vdash e : \tau$, where Ξ is an environment of type variables, Γ an environment associating types to variables, e is an expression, and τ is a type. For well-typed expressions e and e' we define contextual refinement, denoted $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau$. As usual, e and e' are contextually equivalent, denoted $\Xi \mid \Gamma \vDash e \approx_{\mathsf{ctx}} e' : \tau$, if e contextually refines e' and vice versa. With this in place, we can explain which contextual refinements and equivalences we prove for STLang. The soundness of these refinements and equivalences means, of course, that one can use them when reasoning about program equivalences.

The contextual refinements and equivalences that we prove for pure computations are given in Figure 42. To simplify the notation, we have omitted environments Ξ and Γ in the refinements and equivalences in the Figure. Moreover, we do not include assumptions on typing of subexpressions in the Figure; precise formal results are stated in §4.4.

Refinement (Neutrality) expresses that a computation of unit type either diverges or produces the unit value.

$$e \preceq_{\mathsf{ctx}} () : \mathbf{1} \qquad \text{(Neutrality)}$$

$$\mathtt{let}\, x = e_2\, \mathtt{in}\, (e_1, x) \approx_{\mathsf{ctx}} (e_1, e_2) : \tau_1 \times \tau_2 \qquad \text{(Commutativity)}$$

$$\mathtt{let}\, x = e\, \mathtt{in}\, (x, x) \approx_{\mathsf{ctx}} (e, e) : \tau \times \tau \qquad \text{(Idempotency)}$$

$$\mathtt{let}\, y = e_1\, \mathtt{in}\, \mathtt{rec}\, f(x) = e_2 \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = \mathtt{let}\, y = e_1\, \mathtt{in}\, e_2 : \tau_1 \to \tau_2 \qquad \text{(Rec hoisting)}$$

$$\mathtt{let}\, y = e_1\, \mathtt{in}\, \Lambda\, e_2 \preceq_{\mathsf{ctx}} \Lambda\, (\mathtt{let}\, y = e_1\, \mathtt{in}\, e_2) : \forall X.\, \tau \qquad (\Lambda \text{ hoisting})$$

$$e \preceq_{\mathsf{ctx}} \mathtt{rec}\, f(x) = (e\ x) : \tau_1 \to \tau_2 \qquad (\eta \text{ expansion for rec})$$

$$e \preceq_{\mathsf{ctx}} \Lambda\, (e\ \_) : \forall X.\, \tau \qquad (\eta \text{ expansion for } \Lambda)$$

$$(\mathtt{rec}\, f(x) = e_1)\ e_2 \preceq_{\mathsf{ctx}} e_1[e_2, (\mathtt{rec}\, f(x) = e_1)/x, f] : \tau \qquad (\beta \text{ reduction for rec})$$

$$(\Lambda\, e)\ \_ \approx_{\mathsf{ctx}} e : \tau[\tau'/X] \qquad (\beta \text{ reduction for } \Lambda)$$

Figure 42: Contextual Refinements and Equivalences for Pure Computations.

The contextual equivalence in (Commutativity) says that the order of evaluation for pure computations does not matter: the computation on the left first

evaluates e_2 and then e_1, on the right we first evaluate e_1 and then e_2. The contextual equivalence in (Idempotency) expresses the idempotency of pure computations: it does not matter whether we evaluate an expression once, as done on the left, or twice, as done on the right. The contextual refinements in (Rec hoisting) and (Λ hoisting) formulate the soundness of λ-hoisting for ordinary recursive functions and for type functions. The contextual refinements (η expansion for rec) and (η expansion for Λ) express η-rules for ordinary recursive functions and for type functions. The contextual refinements (β reduction for rec) and (β reduction for Λ) express the soundness of β-rules for ordinary recursive functions and for type functions.

In addition, we prove the expected contextual equivalences for monadic computations, shown in Figure 43.

$$\texttt{bind}\, e\, \texttt{in}\, (\lambda x.\, \texttt{return}\, x) \approx_{\text{ctx}} e : \mathsf{ST}\ \rho\ \tau \qquad \text{(Left Identity)}$$

$$e_2\ e_1 \preceq_{\text{ctx}} \texttt{bind}\, (\texttt{return}\, e_1)\, \texttt{in}\, e_2 : \mathsf{ST}\ \rho\ \tau \qquad \text{(Right Identity)}$$

$$\texttt{bind}\, (\texttt{bind}\, e_1\, \texttt{in}\, e_2)\, \texttt{in}\, e_3 \preceq_{\text{ctx}} \texttt{bind}\, e_1\, \texttt{in}\, (\lambda x.\, \texttt{bind}\, (e_2\ x)\, \texttt{in}\, e_3) : \mathsf{ST}\ \rho\ \tau' \qquad \text{(Associativity)}$$

Figure 43: Contextual Equivalences for Stateful Computations.

The results in Figure 42 are the kind of results one would expect for pure computations in a call-by-value language; the challenge is, of course, to show that they hold in the full STLang language, that is, also when subexpressions may involve arbitrary (possibly nested) stateful computations encapsulated using `runST`. That is the purpose of our logical relation, which we present in §4.3. We further use our logical relation to show the following State-Independence theorem:

Theorem 4.1.1 (State Independence)**.**

$$\cdot \mid x : \mathit{STRef}\ \rho\ \tau' \vdash e : \tau \wedge (\exists h_1, \ell, h_2, v.\ \langle h_1, e[\ell/x]\rangle \to^* \langle h_2, v\rangle) \implies \forall h_1', \ell'.\ \exists h_2', v'.\ \langle h_1', e[\ell'/x]\rangle \to^* \langle h_2', v'\rangle \wedge h_1' \subseteq h_2'.$$

This theorem expresses that, if the execution of a well-typed expression e, when x is substituted by some location, in *some* heap h_1 terminates, then running e, when x is substituted by *any* location, in *any* heap h_1' will also terminate in some heap h_2' which is an extension of h_1', *i.e.*, the execution cannot have modified h_1' but it can have allocated new state, via encapsulated stateful computations. Note that this implies that e never reads from or writes to x.

Summary of contributions To sum up, the main contributions of this paper are as follows:

- We present a logical relation for a programming language STLang featuring a parallel to Haskell's ST monad with a construct, `runST`, to encapsulate stateful computations. We use our logical relation to prove that `runST` provides proper encapsulation of state, by showing (1) that contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of stateful computations encapsulated via `runST` and (2) that the State-Independence theorem holds. This is the first time that these results have been established for a programming language with an operational semantics that uses a single global higher-order heap with in-place destructive updates.
- We define our logical relation in Iris, a state-of-the-art higher-order separation logic designed for program verification, using a new approach involving novel predicates defined in Iris, which we explain in §4.5.
- We have formalized the whole technical development, including all proofs of the equations above and the State-Independence theorem, in the Iris implementation in Coq.

The paper is organized as follows. We begin by formally defining STLang, its semantics and typing rules in §4.2. There, we also formally state our definition of contextual refinement and contextual equivalence. In §4.3, we present our logical relation after briefly introducing the parts of Iris needed for a conceptual understanding of the logical relation. We devote §4.4 to the precise statement and proof sketches of the refinements in Figure 42 and Figure 43. In §4.5, we recall some further concepts of Iris and explain how they are used to give a complete technical definition of the logical relation. Readers only interested in the ideas behind the logical relation can skip this section. We describe our formalization of the technical development in the Iris implementation in Coq in §4.6. We discuss related work in §4.7 and conclude in §4.8.

4.2 The STLang language

In this section, we present STLang, a higher-order functional programming language with impredicative polymorphism, recursive types, higher-order store and a ST-like type.

Syntax The syntax of STLang is mostly standard and presented in Figure 44. Note that there are no types in the terms; following [2] we write $\Lambda\, e$ for type abstraction and $e\ \_$ for type application / instantiation. For the stateful part of the language, we use `return` and `bind` for the return and bind operations of the ST monad, and `ref`(e) creates a new reference, `!e` reads from one and `e` $\leftarrow$ `e` writes into one. Finally, `runST` runs effectful computations. Note that

we treat the stateful operations as constructs in the language rather than as special constants.

$$
\begin{aligned}
\circledcirc &::= + \mid - \mid * \mid = \mid < \\
e &::= x \mid () \mid \texttt{true} \mid \texttt{false} \mid n \mid \ell \mid (e, e) \mid \texttt{inj}_i\, e \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, e \\
&\quad \mid \texttt{unfold}\, e \mid e\ e \mid e\ \_ \mid \pi_i\, e \mid \texttt{match}\, e\, \texttt{with}\, \texttt{inj}_i\, x \Rightarrow e_i\, \texttt{end} \\
&\quad \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid e \circledcirc e \mid \texttt{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid e == e \mid \texttt{bind}\, e\, \texttt{in}\, e \\
&\quad \mid \texttt{return}\, e \mid \texttt{runST}\ \{e\} \\
v &::= () \mid \texttt{true} \mid \texttt{false} \mid n \mid \ell \mid (v, v) \mid \texttt{inj}_i\, v \mid \texttt{rec}\, f(x) = e \mid \Lambda\, e \mid \texttt{fold}\, v \\
&\quad \mid \texttt{ref}(v) \mid\ !\, v \mid v \leftarrow v \mid \texttt{bind}\, v\, \texttt{in}\, v \mid \texttt{return}\, v \\
\tau &::= X \mid \rho \mid \mathbf{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \mid \forall X.\, \tau \mid \mu X.\, \tau \\
&\quad \mid \texttt{ref}(\tau) \mid \texttt{ST}\ \rho\ \tau
\end{aligned}
$$

Figure 44: The syntax of STLang.

Typing Typing judgments are of the form $\Xi \mid \Gamma \vdash e : \tau$, where Ξ is a set of type variables, and Γ is a finite partial function from variables to types. An excerpt of the typing rules are shown in Figure 45.

Operational semantics We present a small-step call-by-value operational semantics for STLang, using a transition system $\langle h, e\rangle \to \langle h', e'\rangle$ whose nodes are configurations consisting of a heap h and an expression e. A heap $h \in \mathit{Loc} \rightharpoonup^{\text{fin}} \mathit{Val}$ is a finite partial function that associates values to locations, which we suppose are positive integers ($\mathit{Loc} \triangleq \mathbb{Z}^+$)[2].

The semantics, shown in Figure 46, is presented in the Felleisen-Hieb style [24], using evaluation contexts C: the reduction relation $\to$ is the closure by evaluation context of the *head* reduction relation $\to_h$. Notice that even the "pure" reductions steps, such as β-reduction, mention the heap. The more subtle part of the operational semantics is how the ST monad is handled, indeed, we only want the stateful computations to run when they are wrapped inside `runST`. This is why we define an auxiliary reduction relation, $\langle h, e\rangle \rightsquigarrow \langle h', e'\rangle$. This auxiliary relation is also defined using a head reduction and evaluation contexts $\mathbb{K}$, which are distinct from the evaluation contexts for the main reduction relation. This auxiliary relation in "embedded" in the main one by the rule

$$\frac{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}{\langle h, \texttt{runST}\ \{v\}\rangle \to_h \langle h', \texttt{runST}\ \{e\}\rangle}$$

[2]This choice is due to the fact that Iris library in Coq provides extensive support for the type of positive integers.

$\boxed{\Xi \mid \Gamma \vdash e : \tau}$

$$\textsc{Tvar}\ \frac{}{\Xi \mid \Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Trec}\ \frac{\Xi \mid \Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2}{\Xi \mid \Gamma \vdash \texttt{rec}\, f(x) = e : \tau_1 \to \tau_2} \qquad \textsc{Tabs}\ \frac{\Xi, X \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda e : \forall X.\, \tau}$$

$$\textsc{Tfold}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau / X]}{\Xi \mid \Gamma \vdash \texttt{fold}\, e : \mu X.\, \tau} \qquad \textsc{Tinst}\ \frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]}$$

$$\textsc{Tnew}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \texttt{ref}(e) : \mathsf{ST}\ \rho\ (\mathsf{STRef}\ \rho\ \tau)} \qquad \textsc{Tderef}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash\, !e : \mathsf{ST}\ \rho\ \tau}$$

$$\textsc{Tgets}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : \mathsf{ST}\ \rho\ \mathbf{1}}$$

$$\textsc{Trefeq}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \mathsf{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash e == e' : \mathbb{B}} \qquad \textsc{Treturn}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \texttt{return}\, e : \mathsf{ST}\ \rho\ \tau}$$

$$\textsc{Tbind}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{ST}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau \to (\mathsf{ST}\ \rho\ \tau')}{\Xi \mid \Gamma \vdash \texttt{bind}\, e\, \texttt{in}\, e' : \mathsf{ST}\ \rho\ \tau'}$$

$$\textsc{Trunst}\ \frac{\Xi, X \mid \Gamma \vdash e : \mathsf{ST}\ X\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \texttt{runST}\ \{e\} : \tau}$$

Figure 45: An excerpt of the typing rules for STLang.

Reduction $\boxed{\langle h, e\rangle \to \langle h', e'\rangle}$ and head step $\boxed{\langle h, e\rangle \to_h \langle h', e'\rangle}$

Evaluation contexts:

$$\begin{aligned} C ::= & [\,] \mid (C, e) \mid (v, C) \mid \mathtt{inj}_i\, C \mid \mathtt{fold}\, C \mid \mathtt{unfold}\, C \mid C\ e \mid v\ C \mid C\ \_ \\ & \mid C \circledcirc e \mid v \circledcirc C \mid \pi_i\, C \mid \mathtt{match}\, C\, \mathtt{with}\, \mathtt{inj}_i\, x \Rightarrow e_i\, \mathtt{end} \\ & \mid \mathtt{if}\, C\, \mathtt{then}\, e\, \mathtt{else}\, e \mid \mathtt{ref}(C) \mid\, !\, C \mid C \leftarrow e \mid v \leftarrow C \mid C == e \\ & \mid v == C \mid \mathtt{bind}\, C\, \mathtt{in}\, e \mid \mathtt{bind}\, v\, \mathtt{in}\, C \mid \mathtt{return}\, C \mid \mathtt{runST}\, \{C\} \end{aligned}$$

$$\frac{\langle h, e\rangle \to_h \langle h', e'\rangle}{\langle h, C[e]\rangle \to \langle h', C[e']\rangle} \qquad \langle h, \mathtt{unfold}(\mathtt{fold}\, v)\rangle \to_h \langle h, v\rangle \qquad \langle h, (\Lambda\, e)\ \_\rangle \to_h \langle h, e\rangle$$

$$\langle h, (\mathtt{rec}\, f\, (x) = e)\ v\rangle \to_h \langle h, e[v, \mathtt{rec}\, f\, (x) = e/x, f\,]\rangle \qquad \frac{\ell = \ell'}{\langle h, \ell == \ell'\rangle \to_h \langle h, \mathtt{true}\rangle}$$

$$\langle h, \mathtt{match}\, \mathtt{inj}_i\, v\, \mathtt{with}\, \mathtt{inj}_i\, x \Rightarrow e_i\, \mathtt{end}\rangle \to_h \langle h, e_i[v/x]\rangle$$

$$\frac{\ell \neq \ell'}{\langle h, \ell == \ell'\rangle \to_h \langle h, \mathtt{false}\rangle} \qquad \frac{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}{\langle h, \mathtt{runST}\, \{v\}\rangle \to_h \langle h', \mathtt{runST}\, \{e\}\rangle}$$

$$\langle h, \mathtt{runST}\, \{\mathtt{return}\, v\}\rangle \to_h \langle h, v\rangle$$

Effectful reduction $\boxed{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}$ and head step $\boxed{\langle h, v\rangle \rightsquigarrow_h \langle h', e\rangle}$

Effectful evaluation contexts: $\mathbb{K} ::= [\,] \mid \mathtt{bind}\, \mathbb{K}\, \mathtt{in}\, v$

$$\frac{\langle h, v\rangle \rightsquigarrow_h \langle h', e\rangle}{\langle h, \mathbb{K}[v]\rangle \rightsquigarrow \langle h', \mathbb{K}[e]\rangle} \qquad \langle h, \mathtt{bind}\, (\mathtt{return}\, v)\, \mathtt{in}\, v'\rangle \rightsquigarrow_h \langle h, v'\ v\rangle$$

$$\textsc{Alloc}\quad \frac{\ell \notin \mathrm{dom}(h)}{\langle h, \mathtt{ref}(v)\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathtt{return}\, \ell\rangle}$$

$$\langle h \uplus \{\ell \mapsto v\}, !\, \ell\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathtt{return}\, v\rangle$$

$$\langle h \uplus \{\ell \mapsto v'\}, \ell \leftarrow v\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \mathtt{return}\, ()\rangle$$

If $\rightharpoonup$ is a relation, we note $\rightharpoonup^n$ its iterated self-composition and $\rightharpoonup^*$ its reflexive and transitive closure.

Figure 46: An excerpt of the dynamics of STLang, a call-by-value, small-step operational semantics.

Notice that $\rightsquigarrow$ always reduces *from* a value: this is because values of type `ST` can be seen as "frozen" computations, until they appear inside a `runST`. The expression e on the right hand-side of the rule above can be a reducible expression, which is reduced by using $C = \texttt{runST}\{[]\}$ as a context for the main reduction rule $\rightarrow$.

This operational semantics is new, therefore we include an example of how a simple program reduces. The program initializes a reference r to 3, then writes 7 into r and finally reads r.

$$\langle \emptyset, \texttt{runST}\{\texttt{bind}\,\texttt{ref}(3)\,\texttt{in} \\ \lambda r.\,\texttt{bind}(r \leftarrow 7)\,\texttt{in}\,(\lambda\_.\,\texttt{bind}\,!r\,\texttt{in}\,(\lambda x.\,\texttt{return}\,x))\}\rangle$$

The contents of the `runST` is a value, so we can use the rule above, and the context $\mathbb{K} = \texttt{bind}\,[]\,\texttt{in}\,\cdots$ to reduce $\langle \emptyset, \texttt{ref}(3)\rangle \rightsquigarrow_h \langle [l \mapsto 3], \texttt{return}\,l\rangle$ (for some arbitrary l) and get:

$$\langle [l \mapsto 3], \texttt{runST}\{\texttt{bind}\,(\texttt{return}\,l)\,\texttt{in} \\ \lambda r.\,\texttt{bind}(r \leftarrow 7)\,\texttt{in}\,(\lambda\_.\,\texttt{bind}\,!r\,\texttt{in}\,(\lambda x.\,\texttt{return}\,x))\}\rangle$$

The contents of `runST` is still a value, and this time we use the empty context $\mathbb{K} = []$ and the rule for the `bind` of a `return`,

$$\langle [l \mapsto 3], \texttt{bind}\,(\texttt{return}\,l)\,\texttt{in}\,(\lambda r.\,\cdots)\rangle \rightsquigarrow_h \langle [l \mapsto 3], (\lambda r.\,\cdots)\;l\rangle$$

to get:

$$\langle [l \mapsto 3], \texttt{runST}\,\{(\lambda r.\,\texttt{bind}(r \leftarrow 7)\,\texttt{in}\,(\lambda\_.\,\texttt{bind}\,!r\,\texttt{in}\,(\lambda x.\,\texttt{return}\,x)))\;l\}\rangle$$

This time we use the context $C = \texttt{runST}\,\{[]\}$ and the rule for β-reduction to get:

$$\langle [l \mapsto 3], \texttt{runST}\,\{\texttt{bind}\,(l \leftarrow 7)\,\texttt{in}\,(\lambda\_.\,\texttt{bind}\,!l\,\texttt{in}\,(\lambda x.\,\texttt{return}\,x))\}\rangle$$

The situation is now the same as for the first two reduction steps and we reduce further to:

$$\langle [l \mapsto 7], \texttt{runST}\,\{\texttt{bind}\,(\texttt{return}\,())\,\texttt{in}\,(\lambda\_.\,\texttt{bind}\,!l\,\texttt{in}\,(\lambda x.\,\texttt{return}\,x))\}\rangle$$

and then, in two steps (rule for `bind` and `return`, then β-reduction):

$$\langle [l \mapsto 7], \texttt{runST}\,\{\texttt{bind}\,!l\,\texttt{in}\,(\lambda x.\,\texttt{return}\,x)\}\rangle$$

Finally we get:

$$\langle [l \mapsto 7], \texttt{runST}\,\{\texttt{return}\,7\}\rangle$$

and, from the rule for `runST` and `return v`:

$$\langle [l \mapsto 7], 7\rangle.$$

Having defined the operational semantics and the typing rules we can now define contextual refinement and equivalence. In this definition we write $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; \mathbf{1})$ to express that C is a well-typed closing context (the remaining rules for this relation are completely standard).

Definition 4.2.1 (Contextual refinement and equivalence)**.** *We define contextual refinement* $\preceq_{\mathsf{ctx}}$ *and contextual equivalence* $\approx_{\mathsf{ctx}}$ *as follows.*

$$\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau \triangleq \Xi \mid \Gamma \vdash e : \tau \wedge \Xi \mid \Gamma \vdash e' : \tau \wedge \forall h, h', C.\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; \mathbf{1}) \wedge (h, C[e])\downarrow \implies (h', C[e'])\downarrow$$

$$\Xi \mid \Gamma \vDash e \approx_{\mathsf{ctx}} e' : \tau \triangleq \Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau \wedge \Xi \mid \Gamma \vDash e' \preceq_{\mathsf{ctx}} e : \tau.$$

where $(h, e)\downarrow \triangleq \exists h', v.\, (h, e) \to^* (h', v)$

4.3 Logical Relation

It is well-known that it is challenging to construct logical relations for languages with higher-order store because of the so-called type-world circularity [1, 4, 15]. Other recent work has shown how this challenge can be addressed by using the original Iris logic to define logical relations for languages with higher-order store [39, 42]. In fact, a key point is that Iris has enough logical features to give a direct inductive interpretation of the programming language types into Iris predicates.

The binary relations in [39, 42] were defined using Iris's built-in notion of Hoare triple and weakest precondition. This approach is, however, too abstract for our purposes: to prove the contextual refinements and equivalences for pure computations mentioned in the Introduction, we need to have more fine-grained control over how computations are related.

In this section we start by giving a gentle introduction to the base logic of Iris. Hereafter, we use the Iris base logic to define two new logical connectives called future modality and If-Convergent. We use these, instead of the weakest precondition used in [39, 42], when defining our binary logical relation.

We focus on properties that are necessary for understanding the key ideas of the definition of the logical relation; more technical details, including definitions and lemmas required for proving properties of the logical relation, are deferred until §4.5.

An Iris Primer

Iris was originally presented as a framework for higher-order (concurrent) separation logic, with built-in notions of physical state (in our case heaps), ghost-state (monoids) invariants and weakest preconditions, useful for Hoare-style reasoning about higher-order concurrent imperative programs [33]. Subsequently, Iris was extended with a notion of higher-order ghost state [34],

i.e., the ability to store arbitrary higher-order separation-logic predicates in ghost variables. Recently, a simpler Iris *base logic* was defined, and it was shown how that base logic suffices for defining the earlier built-in concepts of invariants, weakest preconditions, and higher-order ghost state [38].

In Iris one can quantify over the Iris types κ:

$$\kappa ::= \mathbf{1} \mid \kappa \times \kappa \mid \kappa \to \kappa \mid \mathit{Expr} \mid \mathit{Val} \mid \mathbb{Z} \mid \mathbb{B} \mid \kappa \xrightarrow{\text{fin}} \kappa \mid \mathsf{finset}(\kappa) \mid \mathit{Monoid} \mid \mathit{Names} \mid \mathit{iProp} \mid \ldots$$

Here *Expr* and *Val* are the types of syntactic expressions and values of STLang, $\mathbb{Z}$ is the type of integers, $\mathbb{B}$ is the type of booleans, $\kappa \xrightarrow{\text{fin}} \kappa$ is the type of partial functions with finite support, $\mathsf{finset}(\kappa)$ is the type of finite sets, *Monoid* is the type of monoids, *Names* is the type of ghost names, and *iProp* is the type of Iris propositions. A basic grammar for Iris propositions P is:

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \mid \triangleright P \mid \mu r.P \mid \Box P \mid \Rightarrow\!\!\!\!\Vdash P$$

The grammar includes the usual connectives of higher-order separation logic ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $*$, $\mathrel{-\!\!*}$, $\forall$ and $\exists$). In this grammar Φ is an Iris predicate, i.e., a term of type $\kappa \to \mathit{iProp}$ (for appropriate κ). The intuition is that the propositions denote sets of resources and, as usual in separation logic, $P * P'$ holds for those resources which can be split into two disjoint parts, with one satisfying P and the other satisfying P'. Likewise, the proposition $P \mathrel{-\!\!*} P'$ describes those resources which satisfy that, if we combine it with a disjoint resource satisfied by P we get a resource satisfied by P'. In addition to these standard connectives there are some other interesting connectives, which we now explain.

The $\triangleright$ is a modality, pronounced "later". It is used to guard recursively defined propositions: $\mu r.P$ is a well-defined guarded-recursive predicate only if r appears under a $\triangleright$ in P. The $\triangleright$ modality is an abstraction of step-indexing [5, 6, 21]. In terms of step-indexing $\triangleright P$ holds if P holds a step later; thence the name. In Iris it can be used to define weakest preconditions and to guard impredicative invariants to avoid self-referential paradoxes [38]. Here we simply use it to take a guarded fixed point when we give the interpretation of recursive types, similarly to what was done in [21]. For any proposition P, we have that $P \vdash \triangleright P$. The later modality commutes with all of the connectives of higher-order separation logic, including quantifiers.

Another modality of the Iris logic is the "persistence" modality ($\Box$). This modality is used in Iris to capture a sublogic of knowledge (as opposed to resources) that obeys standard rules for intuitionistic higher-order logic. We say that P is *persistent* if $P \vdash \Box P$. Intuitively, $\Box P$ holds if P holds without asserting any exclusive ownership. Hence $\Box P$ is a duplicable assertion, i.e., we have $(\Box P) * (\Box P) \dashv\vdash \Box P$, where $\dashv\vdash$ is the logical equivalence of formulas. Hence

persistent propositions are therefore duplicable. The persistence modality is idempotent, $\Box P \vdash \Box\Box P$, and also satisfies $\Box P \vdash P$. It (and by extension persistence) also commutes with all of the connectives of higher-order separation logic, including quantifiers.

The final modality we present in this section is the "update" modality[3] ($\Rightarrow$). Intuitively, the proposition $\Rightarrow P$ holds for resources that can be updated (through allocation, deallocation, or alteration) to resources that satisfy P, without violating the environment's knowledge or ownership of resources. We write $P \Rrightarrow Q$ as a shorthand for $P \mathrel{-\!\!*} \Rightarrow Q$. The update modality is idempotent, $\Rightarrow(\Rightarrow P) \dashv\vdash \Rightarrow P$.

Future Modality and If-Convergent

In this subsection we define two new constructs in Iris, which we will use to define the logical relation. The first construct, the future modality, will allow us to reason about what will happen in a "future world". The second construct, the If-Convergent predicate, will be used instead of weakest preconditions to reason about properties of computations.

Future Modality We define the *future modality* $\Rightarrow\!\!>\!\!\{\cdot\}\!\!\Rightarrow\!\!\triangleright$ as follows:

$$\Rightarrow\!\!>\!\!\{n\}\!\!\Rightarrow\!\!\triangleright P \triangleq (\Rightarrow \triangleright)^n \Rightarrow P$$

where $(\Rightarrow \triangleright)^n$ is n times repetition of $\Rightarrow \triangleright$. Intuitively, $\Rightarrow\!\!>\!\!\{n\}\!\!\Rightarrow\!\!\triangleright P$ expresses that n steps into the future, we can update our resources to satisfy P. We write $P \Rrightarrow\!\!>\!\!\{n\}\!\!\Rrightarrow Q$ as a shorthand for $P \mathrel{-\!\!*} \Rightarrow\!\!>\!\!\{n\}\!\!\Rightarrow\!\!\triangleright Q$.

If-Convergent (IC) We define the *If-Convergent (IC)* predicate in Iris as follows:

$$\mathsf{IC}^{\gamma}\, e \{\!| v.\ Q |\!\} \triangleq$$
$$\forall h_1, h_2, v, n.\ \langle h_1, e\rangle \rightarrow^n \langle h_2, v\rangle * \mathit{heap}_{\gamma}(h_1) \Rrightarrow\!\!>\!\!\{n\}\!\!\Rrightarrow \mathit{heap}_{\gamma}(h_2) * Q\ v$$

In general the number of steps, n, can also appear in Q but here we only present this slightly simpler version. The $\mathsf{IC}^{\gamma}\, e \{\!| v.\ Q |\!\}$ predicate expresses that, for any heap h_1, if (e, h_1) can reduce to (v, h_2) in n steps, and if we have ownership over h_1, then, n steps into the future, we will have ownership over the heap h_2, and the postcondition Q will hold.

A crucial feature of the IC predicate is that it allows us to use a ghost state name γ to keep track of the contents of the heap during the execution of e. This allows us to abstract away from the concrete heaps when reasoning about IC predicates[4]. Note that the IC predicate does *not* require that it is *safe* to

[3] In [38] this modality is called the *fancy* update modality. Technically, this modality comes equipped with certain "masks" but we do not discuss those here.

[4] This is related to the way the definition of weakest preconditions in Iris hides state [38].

execute the expression e: in particular, if e gets stuck (or diverges) in all heaps, then $\mathsf{IC}^{\gamma}\, e \{\!| v.\ Q |\!\}$ holds trivially.

The predicate $heap_{\gamma}(h_1)$ is a ghost state predicate stating ownership of a logical heap identified by the ghost state name γ (one can think of this as the usual ownership of a heap in separation logic). Ownership of a logical heap cell l is written as $\ell \mapsto_{\gamma} v$, and says that the heap identified by γ stores the value v at location ℓ. We show the precise definition of $heap_{\gamma_h}(h)$ and $\ell \mapsto_{\gamma} v$ in §4.5; here we just highlight the properties that these abstract predicates enjoy:

$$heap_{\gamma}(h) * \ell \mapsto_{\gamma} v \Rrightarrow h(l) = v \tag{4.1}$$

$$heap_{\gamma}(h) \wedge l \notin \mathrm{dom}(h) \ \equiv\!\!\!\!* \ heap_{\gamma}(h[l \mapsto v]) * \ell \mapsto_{\gamma} v \tag{4.2}$$

$$heap_{\gamma}(h) * \ell \mapsto_{\gamma} v \ \equiv\!\!\!\!* \ heap_{\gamma}(h[l \mapsto v']) * \ell \mapsto_{\gamma} v' \tag{4.3}$$

$$\ell \mapsto_{\gamma} v * \ell \mapsto_{\gamma} v' \ \Rrightarrow \bot \tag{4.4}$$

Property (4.1) says that if we have ownership of a heap h and a location l pointing to v, both with the same ghost name γ, then we know that $h(l) = v$. Property (4.2) expresses that we can allocate a new location l in h, if l is not already in the domain of h. Finally, Property (4.3) says that we can update the value at location l, if we have both $heap_{\gamma}(h)$ and $\ell \mapsto_{\gamma} v$. Property (4.4) expresses exclusivity of the ownership of locations.

Akin to the way Hoare triples are defined in Iris using the weakest precondition, we define a new notion called IC triple as follows:

$$\{\!| P |\!\}\, e \{\!| v.\ Q |\!\}_{\gamma} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{IC}^{\gamma}\, e \{\!| v.\ Q |\!\})$$

The IC triple says, that given resources described by P, if e reduces in a heap identified by γ, then the post-condition Q will hold. Notice that the IC triple is a persistent predicate and is not allowed to own any exclusive resources.

Definition of the Logical Relation

We now have enough logical machinery to describe the logical relation (pedantically, it is a family of logical relations) shown in Figure 46. The logical relation is a binary relation, which allows us to relate pairs of expressions and pairs of values to each other. We will show that if two expressions are related in the logical relation, then the left hand side expression contextually approximates the right hand side expression. Therefore, we sometimes refer to the the left hand side as the implementation and the right hand side as the specification.

The value relation $[\![\Xi \vdash \tau]\!]_{\Delta}$ is an Iris relation of type $(Val \times Val) \to iProp$ and, intuitively, it relates STLang values of type τ. The value relation is defined by induction on the type τ. Here, Ξ is an environment of type variables, and Δ is a semantic environment for these type variables, as is usual for languages with parametric polymorphism [65].

Value relations:

$$\begin{aligned}
[\![\Xi \vdash X]\!]_\Delta &\triangleq (\Delta(X)).1 \\
[\![\Xi \vdash \mathbf{1}]\!]_\Delta(v, v') &\triangleq v = v' = () \\
[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v, v') &\triangleq v = v' \in \{\texttt{true}, \texttt{false}\} \\
[\![\Xi \vdash \mathbb{Z}]\!]_\Delta(v, v') &\triangleq v = v' \in \mathbb{Z} \\
[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v, v') &\triangleq \exists w_1, w_2, w_1', w_2'.\ v = (w_1, w_2) \wedge v' = (w_1', w_2') \wedge \\
&\quad [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') \wedge [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2') \\
[\![\Xi \vdash \tau + \tau']\!]_\Delta(v, v') &\triangleq (\exists w, w'.\ v = \texttt{inj}_1\, w \wedge v' = \texttt{inj}_1\, w' \wedge \\
&\quad [\![\Xi \vdash \tau]\!]_\Delta(w, w')) \vee (\exists w, w'.\ v = \texttt{inj}_2\, w \wedge \\
&\quad v' = \texttt{inj}_2\, w' \wedge [\![\Xi \vdash \tau']\!]_\Delta(w, w')) \\
[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') &\triangleq \Box\Big(\forall (w, w').\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow \\
&\quad \mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(v\ w, v'\ w')\Big) \\
[\![\Xi \vdash \forall X.\, \tau]\!]_\Delta(v, v') &\triangleq \Box\Big(\forall f, r \in \mathcal{R}.\ \mathsf{persistent}(f) \Rightarrow \\
&\quad \mathcal{E}\,[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f,r)}\,(v\ _, v'\ _)\Big) \\
[\![\Xi \vdash \mu X.\, \tau]\!]_\Delta(v, v') &\triangleq \mu f.\ \exists w, w'.\ v = \texttt{fold}\, w \wedge v' = \texttt{fold}\, w' \wedge \\
&\quad \triangleright [\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f, \mathsf{toRgn}(\Delta, \mu X.\, \tau))}(w, w') \\
[\![\Xi \vdash \mathtt{STRef}\ \rho\ \tau]\!]_\Delta(v, v') &\triangleq \exists \ell, \ell', r.\ v = \ell \wedge v' = \ell' \wedge \mathsf{isRgn}(\mathsf{toRgn}(\Delta, \rho), r) * \\
&\quad \mathsf{bij}(r, \ell, \ell') * \mathsf{rel}(r, \ell, \ell', [\![\Xi \vdash \tau]\!]_\Delta) \\
[\![\Xi \vdash \mathtt{ST}\ \rho\ \tau]\!]_\Delta(v, v') &\triangleq \forall \gamma_h, \gamma_h', h_1'. \\
&\quad \Big\{ heap_{\gamma_h'}(h_1') * \mathsf{regions} * \mathsf{region}(\mathsf{toRgn}(\Delta, \rho), \gamma_h, \gamma_h') \Big\} \\
&\quad \texttt{runST}\ \{v\} \\
&\quad \Big\{ w.\ (h_1', \texttt{runST}\ \{v'\}) \Downarrow^{\gamma_h'}_{[\![\Xi \vdash \tau]\!]_\Delta(w, \cdot)} * \mathsf{region}(\mathsf{toRgn}(\Delta, \rho), \gamma_h, \gamma_h') \Big\}_{\gamma_h}
\end{aligned}$$

Expression relation:

$$\mathcal{E}\,\Phi\,(e, e') \triangleq \forall \gamma_h, \gamma_h', h_1'.\ \Big\{ heap_{\gamma_h'}(h_1') * \mathsf{regions} \Big\}\, e \,\Big\{ w.\ (h_1', e') \Downarrow^{\gamma_h'}_{\Phi(w, \cdot)} \Big\}_{\gamma_h}$$

Environment relation:

$$\begin{aligned}
\mathcal{G}[\![\Xi \vdash \cdot]\!]_\Delta(\vec{v}, \vec{v'}) &\triangleq \top \\
\mathcal{G}[\![\Xi \vdash \Gamma, x : \tau]\!]_\Delta(w\vec{v}, w'\vec{v'}) &\triangleq [\![\Xi \vdash \tau]\!]_\Delta(w, w') * \mathcal{G}[\![\Xi \vdash \Gamma]\!]_\Delta(\vec{v}, \vec{v'})
\end{aligned}$$

Logical relatedness:

$$\Xi \mid \Gamma \vDash e \preceq_{\log} e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v'}.\ \mathcal{G}[\![\Xi \vdash \Gamma]\!]_\Delta(\vec{v}, \vec{v'}) \Rightarrow \mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$

$$\mathsf{toRgn}(\Delta, \tau) \triangleq \begin{cases} \Delta(X).2 & \text{if } \tau = X \text{ is a type variable} \\ 1 & \text{otherwise} \end{cases}$$

Figure 46: Binary logical relation.

If τ is a ground type like $\mathbf{1}, \mathbb{B}$ or $\mathbb{Z}$, two values are related at type τ if and only if they are equal (and compatible with the type). For instance, if τ is $\mathbb{Z}$, then $[\![\Xi \vdash \mathbb{Z}]\!]_\Delta(v, v') \triangleq v = v' \in \mathbb{Z}$.

For a product type of the form $\tau \times \tau'$, two values v and v' are related if and only if they both are pairs, and the corresponding components are related at their respective types:

$$[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v, v') \triangleq \exists w_1, w_2, w_1', w_2'.\ v = (w_1, w_2) \wedge v' = (w_1', w_2') \wedge [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') \wedge [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')$$

Note that the formula on the right hand side of $\triangleq$ is simply a formula in (the first order fragment of) Iris. The case of sum types is handled in a very similar fashion.

Two values v and v' are related at a function type $\tau \to \tau'$ if, given any two related values w and w' at type τ, the applications $v\ w$ and $v'\ w'$ are related at type τ'. Notice that those latter two terms are expressions, not values; thus they have to be related under the expression relation $\mathcal{E}[\![\Xi \vdash \tau']\!]_\Delta$, which we will define later. Using Iris, the case for function types is defined as follows:

$$[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') \triangleq \Box\Big(\forall (w, w').\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow \mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(v\ w, v'\ w')\Big)$$

The $\Box$ modality is used to ensure that $[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v')$ is *persistent* and hence duplicable. In fact, we will make sure that all predicates $[\![\Xi \vdash \tau]\!]_\Delta(v, v')$ are persistent. The intuition behind this is that the types of STLang just express duplicable knowledge (the type system is not a substructural type system involving resources).

Let us now discuss the case of polymorphic types. We use the semantic environment Δ, which maps type variables to pairs consisting of an Iris relation on values (the semantic value relation interpreting the type variable) and a region name (we use positive integers, $\mathbb{Z}^+$, to identify regions):

$$\Delta : \mathit{Tvar} \to (((\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}) \times \mathbb{Z}^+)$$

Thus, we simply define $[\![\Xi \vdash X]\!]_\Delta \triangleq \Delta(X).1$.

For type abstraction, two values v and v' are related at $\forall X.\, \tau$ when $v\ \_$ and $v'\ \_$ are related at τ, where the environments (Ξ and Δ) have been extended

with X, and any persistent binary value relation f. (Recall that $v\_$ is the syntax for type application).

$$[\![\Xi \vdash \forall X.\,\tau]\!]_\Delta(v,v') \triangleq \Box\Big(\forall f.\,\mathsf{persistent}(f) \Rightarrow \mathcal{E}\,[\![\Xi, X \vdash \tau]\!]_{\Delta, X\mapsto f}\,(v\ \_, v'\ \_)\Big)$$

The last case, before we get to the types associated to the ST monad, is the case of recursive types: two values are related at type $\mu X.\,\tau$ if they are of the form $\texttt{fold}\,w$ and $\texttt{fold}\,w'$ and, moreover, w and w' are related at τ, where the type variable X is added to the environments, and mapped in Δ to $([\![\Xi \vdash \mu X.\,\tau]\!]_\Delta, \mathsf{toRgn}(\Delta, \mu X.\,\tau))$ (ignore $\mathsf{toRgn}(\Delta, \mu X.\,\tau)$ for now):

$$[\![\Xi \vdash \mu X.\,\tau]\!]_\Delta(v,v') \triangleq \mu f.\Big(\exists w, w'.\ v = \texttt{fold}\,w \wedge v' = \texttt{fold}\,w' \wedge \\ \triangleright [\![\Xi, X \vdash \tau]\!]_{\Delta, X\mapsto (f, \mathsf{toRgn}(\Delta, \mu X.\,\tau))}\,(w,w')\Big)$$

Notice that we use a guarded recursive predicate in Iris, which is well-defined because the occurrence of f is guarded by the later modality $\triangleright$.

Before describing the cases for $\texttt{STRef}\ \rho\ \tau$ and $\texttt{ST}\ \rho\ \tau$ we touch upon the expression relation, which is defined independently of the value relation and has the following type:

$$\mathcal{E}\,\cdot : ((\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}) \to (\mathit{Expr} \times \mathit{Expr}) \to \mathit{iProp}$$

Intuitively, the expression relation $\mathcal{E}\,\Phi\,(e,e')$ holds for two expressions e and e' if e (the implementation) refines, or approximates, e' (the specification). That is, reduction steps taken by e can be simulated by zero or more steps in e'. We use IC triples to define the expression relation. The IC triples are unary and are used to express a property of the implementation expression e. We use the following Iris assertion in the postcondition of the IC triple to talk about the reductions in the specification expression e':

$$(h_1', e') \Downarrow_\Phi^\gamma \ \triangleq\ \exists h_2', v'.\ \langle h_1', e'\rangle \to_d^* \langle h_2', v'\rangle * \mathit{heap}_\gamma(h_2') * \Phi(v')$$

This assertion says that there exists a *deterministic* reduction from (h_1', e') to (h_2', v'), that the resulting heap h_2' is owned and the value satisfies Φ. The deterministic reduction relations, $\to_d$ and $\rightsquigarrow_d$, are defined by the same inference rules as $\to$ and $\rightsquigarrow$, except that the only non-deterministic rule, Alloc, is replaced by a deterministic one:

$$\textsc{det-Alloc}\ \frac{\ell = \mathit{min}(\mathit{Loc} \setminus \mathrm{dom}(h))}{\langle h, \texttt{ref}(v)\rangle \rightsquigarrow_h \langle h \uplus \{\ell \mapsto v\}, \texttt{return}\,\ell\rangle}$$

The requirement that the reduction on the specification side is *deterministic* is used crucially in the proofs of the purity properties in §4.4. We emphasize

that even with this requirement, we can still prove that logical relatedness implies contextual refinement (without requiring that STLang use deterministic reductions), essentially since we only require determinism on the specification side.

Thus, in more detail, the expression relation $\mathcal{E}\ \Phi\ (e, e')$ says that, when given full ownership of a heap h_1' for the specification side ($heap_{\gamma'}(h_1')$), if e reduces to a value w when given some heap h (quantified in IC), then a deterministic reduction on the specification side exists, and the resulting values are related. Notice that the heaps used for the implementation and specification side reductions are universally quantified, because we quantify over the ghost names γ_h, and γ_h', and that we do not require any explicit relationship between them. The persistent Iris assertion regions is responsible for keeping track of all allocated regions; it will be explained later.

For the value interpretation of $\texttt{STRef}\ \rho\ \tau$ and $\texttt{ST}\ \rho\ \tau$, the key idea is to tie each type ρ in an ST monad type (ρ in $\texttt{ST}\ \rho\ \tau$) to a semantic region name $r \in \mathbb{Z}^+$. The association can be looked up using the function toRgn. Intuitively, a region r contains a collection of pairs of locations (one for the implementation side and one for the specification side) in one-to-one correspondence, together with a semantic predicate ϕ for each pair of locations in the region. The idea is that an implementation-side heap h and a specification-side heap h' satisfies a region r if, for any pair of locations (ℓ, ℓ') in r, we have values v and v', such that $h(\ell) = v$ and $h'(\ell') = v'$ and $\phi(v, v')$. All this information is contained in the predicate $\mathsf{region}(r, \gamma_h, \gamma_h')$, where γ_h and γ_h' are the ghost names for the implementation and specification heap, respectively.

We have to maintain a one-to-one correspondence between locations because the operational semantics allows for comparison of locations. Given the one-to-one correspondence, we know that two locations on the implementation side are equal *if and only if* their two related counterparts on the specification side are.

We write $\mathsf{isRgn}(r, \rho)$ to say that r is the semantic region tied to ρ. We keep track of all regions by the regions assertion, which allows us to allocate new regions, as so:

$$\mathsf{regions} \Rrightarrow \exists r.\ \mathsf{region}(r, \gamma_h, \gamma_h') \tag{4.5}$$

Notice that (4.5) gives back a fresh semantic region r. The $\mathsf{region}(r, \gamma_h, \gamma_h')$ predicate allows for local reasoning about relatedness of two locations in a region r. We use a predicate $\mathsf{bij}(r, \ell, \ell')$, which in conjunction with region captures that ℓ and ℓ' are related by the one-to-one correspondence in r. Similarly, we use a predicate $\mathsf{rel}(r, \ell, \ell', \phi)$ in conjunction with region for local reasoning about the fact that values at locations ℓ and ℓ' in region r are related by predicate ϕ.

With this in mind, the definition of the value relation for $\texttt{STRef}\ \rho\ \tau$ is that there exists a semantic region r and locations ℓ and ℓ' in a bijection,

$\mathsf{bij}(r,\ell,\ell')$, such that values pointed to by these locations are related by the relation corresponding to the type τ, asserted by $\mathsf{rel}(r,\ell,\ell',[\![\Xi \vdash \tau]\!]_\Delta)$.

Finally, (v,v') are related by $[\![\Xi \vdash \mathsf{ST}\ \rho\ \tau]\!]_\Delta$ if, for any h_1 and h_1' related in r ($\mathsf{region}(r,\gamma_h,\gamma_h')$) along with some h_2 and w such that $\langle h_1, \mathtt{runST}\ \{v\}\rangle \to^* \langle h_2, w\rangle$, then there is a heap h_2' and a value w' such that we *afterwards* have $\left\langle h_1', \mathtt{runST}\ \{v'\}\right\rangle \to_d^* \left\langle h_2', w'\right\rangle$ and $\mathsf{region}(r,\gamma_h,\gamma_h')$ still holds. The intuitive meaning of the word *afterwards* refers to an application of the future modality (in the IC triple). Note that it is important that the semantic region r still holds after $\mathtt{runST}\ \{v\}$ and $\mathtt{runST}\ \{v'\}$ have been evaluated. This captures that encapsulated computations cannot modify the values of existing locations, but may allocate new locations (in new regions).

We have now completed the explanation of the value and expression relation for closed values and expressions. As usual for logical relations, we then relate open terms by closing them by related substitutions, as specified according the environment relation $\mathcal{G}$, and finally relate them in the expression relation for closed terms, see the definition of $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e' : \tau$ in Figure 46.

Properties of the Logical Relation

To show the fundamental theorem and the soundness of the logical relation wrt. contextual approximation, we prove compatibility lemmas for all typing rules. Instead of working with the explicit definition of the IC triple, we make use of the following properties of IC:

Lemma 4.3.1 (Properties of IC)**.**

1. $IC^\gamma\ e\ \{\!|v.\ Q|\!\} * (\forall w.\ (Q\ w) \mathrel{-\!\!*} IC^\gamma\ C[w]\ \{\!|v.\ Q'\ v|\!\}) \vdash IC^\gamma\ C[e]\ \{\!|v.\ Q'|\!\}$
2. $\models\!\!\Rrightarrow (Q\ w) \vdash IC^\gamma\ w\ \{\!|v.\ Q|\!\}$
3. $(\forall v.\ (P\ v) \mathrel{\mathrlap{-}\!\!\Rrightarrow} (Q\ v)) * IC^\gamma\ e\ \{\!|v.\ P|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$
4. $\models\!\!\Rrightarrow IC^\gamma\ e\ \{\!|v.\ Q|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$
5. $IC^\gamma\ e\ \{\!|v.\ \models\!\!\Rrightarrow Q|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$
6. $(\forall h.\ \langle h,e\rangle \to \langle h,e'\rangle) * \triangleright IC^\gamma\ e'\ \{\!|v.\ Q|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$
7. $\triangleright(\forall \ell.\ \ell \mapsto_\gamma v \mathrel{\mathrlap{-}\!\!\Rrightarrow} Q\ \ell) \vdash IC^\gamma\ \mathtt{runST}\ \{\mathtt{ref}(v)\}\ \{\!|w.\ Q|\!\}$
8. $\triangleright \ell \mapsto_\gamma v * \triangleright(\ell \mapsto_\gamma v \mathrel{\mathrlap{-}\!\!\Rrightarrow} Q\ v) \vdash IC^\gamma\ \mathtt{runST}\ \{!\ell\}\ \{\!|w.\ Q|\!\}$
9. $\triangleright \ell \mapsto_\gamma v' * \triangleright(\ell \mapsto_\gamma v \mathrel{\mathrlap{-}\!\!\Rrightarrow} Q\ ()) \vdash IC^\gamma\ \mathtt{runST}\ \{\ell \leftarrow v\}\ \{\!|w.\ Q|\!\}$
10. $IC^\gamma\ \mathtt{runST}\ \{e\}\ \{\!|v.\ Q|\!\} * \Big(\forall w.\ (Q\ w) \mathrel{-\!\!*} IC^\gamma\ \mathtt{runST}\ \{\mathbb{K}[\mathtt{return}\,w]\}\ \{\!|v.\ Q'\ w|\!\}\Big) \vdash IC^\gamma\ \mathtt{runST}\ \{\mathbb{K}[e]\}\ \{\!|v.\ Q'|\!\}$

Items (1) and (2) above show that IC is a monad in the same way that weakest precondition is a monad, known as the Dijkstra monad. Item (3) allows one to strengthen the post-condition. Items (4) and (5) says that we can dispense with the update modality $\Rrightarrow$ for IC since the update modality is idempotent and IC is based on the update modality. Item (6) says that if a pure reduction from e to e' exists and later the postcondition Q will hold when reducing e', then Q will also hold when reducing e. Items (7),(8) and (9) are properties that allow to allocate, read and modify the heap, all expressing, that the post-condition Q will hold, if the resources needed are given and Q holds for the updated resources. Finally, (10) captures the "bind" property for the RunST monad.

All the compatibility lemmas have been proved in the Coq formalization; here we just sketch the proof of the compatibility lemma for runST:

Lemma 4.3.2 (Compatibility for runST)**.** *Suppose* $\Xi, X \mid \Gamma \vDash e \preceq_{\log} e' : ST\ X\ \tau$ *and* $\Xi \vdash \tau$*. Then*

$$\Xi \mid \Gamma \vDash runST\,\{e\} \preceq_{\log} runST\,\{e'\} : \tau$$

Proof Sketch. We prove that for any f and r that $[\![\Xi, X \vdash \mathsf{ST}\ X\ \tau]\!]_{\Delta, X \mapsto (f,r)}(v, v')$ implies $\mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(\mathtt{runST}\,\{v\}, \mathtt{runST}\,\{v'\})$. The lemma follows from the assumption that e and e' are suitably related. Assume we have regions, ghost names for the implementation and specification side, γ_h and γ_h', and $heap_{\gamma_h'}(h_1')$ for some h_1'. We are to show:

$$\mathsf{IC}^{\gamma_h}\,\mathtt{runST}\,\{v\}\left\{\left|\begin{array}{r} w.\ \exists h_2', w'.\ \langle h_1', \mathtt{runST}\,\{v'\}\rangle \to_d^* \langle h_2', w'\rangle * \\ heap_{\gamma_h'}(h_2') * [\![\Xi \vdash \tau]\!]_\Delta(w, w') \end{array}\right|\right\}$$

Using (4.5) with regions we know there exists a fresh semantic region r and that the predicate $\mathsf{region}(r, \gamma_h, \gamma_h')$ holds for r. We then instantiate our assumption by the unit relation $[\![\Xi \vdash \mathbf{1}]\!]_\Delta$ and r to get $[\![\Xi, X \vdash \mathsf{ST}\ X\ \tau]\!]_{\Delta, X \mapsto ([\![\Xi \vdash \mathbf{1}]\!]_\Delta, r)}(v, v')$.

By the definition of the value relation for the type $\mathsf{ST}\ X\ \tau$, we get that if we give a starting specification heap $heap_{\gamma_h'}(h_1')$ and $\mathsf{region}(r, \gamma_h, \gamma_h')$, then we have $\mathtt{runST}\,\{v\}$ reduces to a value w, and there exist a reduction on the specification side producing w' such that w and w' are related by $[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto ([\![\Xi \vdash \mathbf{1}]\!], \rho)}$. Moreover, we also get the ownership of the resulting specification heap $heap_{\gamma_h}(h_2')$.

By Lemma 4.3.1 (3), it suffices to show: $\Rrightarrow \exists h_2', w'.\ \left\langle h_1', \mathtt{runST}\,\{v'\}\right\rangle \to_d^* \left\langle h_2', w'\right\rangle *$ $heap_{\gamma_h'}(h_2') * [\![\Xi \vdash \tau]\!]_\Delta(w, w')$. The only thing that we do not immediately have from our assumption is $[\![\Xi \vdash \tau]\!]_\Delta(w, w')$, we only have that w and w' are related in a larger environment. However since X does not appear free in τ (which follows from $\Xi \vdash \tau$) it follows by induction on τ that $[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto ([\![\Xi \vdash \mathbf{1}]\!], r)}(w, w') \dashv\vdash [\![\Xi \vdash \tau]\!]_\Delta(w, w')$ which concludes the proof. □

Notice that in the above proof we start out with two completely unrelated heaps for the specification and the implementation side since these are universally quantified inside the IC triple. We then establish a *trivial* relation

between them by creating a new *empty* region. We extend and maintain this relation during the simulation of the stateful expressions on both sides. This is in essence the reason why our expression relations need not assume (or guarantee at the end) any relation between the heaps on the implementation and specification sides.

Using the compatibility lemmas, we can prove the following two theorems.

Theorem 4.3.3 (Fundamental theorem)**.** $\Xi \mid \Gamma \vdash e : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : \tau$

Theorem 4.3.4 (Soundness of logical relation)**.**

$$\Xi \mid \Gamma \vdash e : \tau \wedge \Xi \mid \Gamma \vdash e' : \tau \wedge \Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e' : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} e' : \tau$$

4.4 Proving Contextual Refinements and Equivalences

In this section we show how to prove the contextual refinements and equivalences mentioned in the Introduction. For the sake of illustration we present the proofs of Neutrality and one side of the Commutativity theorems in moderate detail — the proofs of these two cases demonstrate the key techniques that are also used to show the remaining contextual refinements and equivalences from the Introduction. For the remaining theorems, we only sketch their proofs at a higher level of abstraction. Readers who are eager to see all proofs in all their details are thus referred to our Coq formalization.

Theorem 4.4.1 (Neutrality)**.** *If* $\Xi \mid \Gamma \vdash e : \mathbf{1}$ *then* $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{ctx}} () : \mathbf{1}$

Proof Sketch. By the fundamental theorem we have $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : \mathbf{1}$. We show that this implies $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} () : \mathbf{1}$. The final result follows from the soundness theorem.

By unfolding the IC predicate, we get the assumption that $\langle h_1, e\rangle \to^* \langle h_2, v\rangle$, including the ownership of $heap_{\gamma_h}(h_1)$ and $heap_{\gamma'_h}(h'_1)$, and have to prove that[5] $\left\langle h'_1, ()\right\rangle$ reduces deterministically to a value w (and some heap) and that (v, w) are in the value relation for the unit type. We proceed by allocating a copy of h'_1, obtaining $heap_{\gamma}(h'_1)$ for some fresh γ. We use this together with our assumptions, notably $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : \mathbf{1}$, to get that $\left\langle h'_1, e\right\rangle \to^*_d \left\langle h'_2, v'\right\rangle$ for some v' and h'_2 such that (v, v') are related in the value relation for the unit type, i.e., $v = v' = ()$, $heap_{\gamma_h}(h_2)$ and $heap_{\gamma}(h'_2)$. Notice that we have, crucially, retained the ownership of $heap_{\gamma'_h}(h'_1)$ and have only updated the freshly allocated copy of h'_1 with the fresh name γ. We are allowed to do this because the relatedness of expressions, as in $\Xi \mid \Gamma \vDash e \preceq_{\mathsf{log}} e : \mathbf{1}$, universally quantifies over ghost names for the specification and implementation side heaps. We conclude the proof by noting that since () is a value, we have, trivially, $\left\langle h'_1, ()\right\rangle \to^*_d \left\langle h'_1, ()\right\rangle$ and that $(v, ())$ are related at the unit type. □

[5] We ignore the future modality for the sake of simplicity.

Theorem 4.4.2 (Commutativity). *If* $\Xi \mid \Gamma \vdash e_1 : \tau_1$ *and* $\Xi \mid \Gamma \vdash e_2 : \tau_2$ *then*

$$\Xi \mid \Gamma \vDash \texttt{let}\, x = e_2\, \texttt{in}(e_1, x) \approx_{\text{ctx}} (e_1, e_2) \;:\; \tau_1 \times \tau_2$$

Proof Sketch. We only show $\Xi \mid \Gamma \vDash \texttt{let}\, x = e_2\, \texttt{in}(e_1, x) \preceq_{\text{log}} (e_1, e_2) \;:\; \tau_1 \times \tau_2$, the other direction is similar. Unfolding the IC predicate we get the assumption that $\langle h_1, \texttt{let}\, x = e_2\, \texttt{in}(e_1, x)\rangle \to^* \langle h_2, v\rangle$ for some h_2 and v, the ownership of $heap_{\gamma_h}(h_1)$ and $heap_{\gamma'_h}(h'_1)$ and we have to prove that $\left\langle h'_1, (e_1, e_2)\right\rangle \to^*_d \left\langle h'_2, v'\right\rangle$ for some h'_2 and v', and that (v, v') are in the value relation for $\tau \times \tau'$. From the first assumption, we can conclude that $\langle h_1, e_2\rangle \to^* \langle h_3, v_2\rangle$, $\langle h_3, e_1\rangle \to^* \langle h_2, v_1\rangle$ and that $v = (v_1, v_2)$.

We proceed by allocating a fresh copy of h_3 (the heap in the middle of execution of the implementation side) with the fresh name γ, $heap_\gamma(h_3)$ and also a fresh $heap_{\gamma'}(h'_1)$ (the heap at the beginning of execution of the specification side). Notice that these are heaps (on either side) immediately before executing e_1. We use these freshly allocated heaps together with $\Xi \mid \Gamma \vDash e_1 \preceq_{\text{log}} e_1 : \tau_1$ (which follows from the fundamental theorem) to conclude[6] $\left\langle h'_1, e_1\right\rangle \to^*_d \left\langle h'_3, v'_1\right\rangle$ for some v'_1 and h'_3.

Now we have the information about the starting heap for execution of e_2 on the specification side. Thus, we are ready to simulate the execution of e_2 on both sides. Note that the order of simulations is dictated by the order on the implementation side as we have to prove that the implementation side is simulated by the specification side.

To simulate e_2 we proceed by allocating a fresh copy of h'_3 (the heap immediately before executing e_2 on the specification side) with a fresh name γ'', $heap_{\gamma''}(h'_3)$. We use this, together with $heap_{\gamma_h}(h_1)$ (which we originally got by unfolding the IC predicate) and $\Xi \mid \Gamma \vDash e_2 \preceq_{\text{log}} e_2 : \tau_2$ (which we know from the fundamental theorem). We can do this as we know $\langle h_1, e_2\rangle \to^* \langle h_3, v_2\rangle$. This allows us to conclude that $\left\langle h'_3, e_2\right\rangle \to^*_d \left\langle h'_2, v'_2\right\rangle$ for some h'_2 and v'_2, the ownership of $heap_{\gamma''}(h'_2)$ and $heap_{\gamma_h}(h_3)$ together with the fact that (v_2, v'_2) are related at type τ_2.

Now we are ready to simulate e_1 on both sides. We use $\Xi \mid \Gamma \vDash e_1 \preceq_{\text{log}} e_1 : \tau_1$ (which we know from the fundamental theorem) together with $heap_{\gamma_h}(h_3)$ (from simulating e_2) and $heap_{\gamma'_h}(h'_1)$ (which we had as an assumption from the definition relatedness). We can do this because we know that $\langle h_3, e_1\rangle \to^* \langle h_2, v_1\rangle$. This allows us to conclude that $\left\langle h'_1, e_1\right\rangle \to^*_d \left\langle h''_3, v''_1\right\rangle$ for some h''_3 and v''_1, the ownership of $heap_{\gamma'_h}(h''_3)$ and $heap_{\gamma_h}(h_2)$ together with the fact that (v_1, v''_1) are related at type τ_1. It follows from the determinism of reduction on the specification side that $h'_3 = h''_3$ and $v'_1 = v''_1$.

The only thing we need to conclude the proof is the ownership of $heap_{\gamma'_h}(h_2)$ (the heap at the end of execution of the specification side) whereas we own $heap_{\gamma'_h}(h''_3)$ which is the heap of the specification side after execution of e_1

[6] For simplicity, we are ignoring some manipulations involving the future modality.

and before execution of e_2. However, using some resource reasoning (which depends on details explained in §4.5), we can conclude that $h_3'' \subseteq h_2$. This in turn allows us to update our heap resource to get $heap_{\gamma_h'}(h_2)$, which concludes the proof. □

The proof sketches of the two theorems above show that the true expressiveness of our logical relation comes from the fact that the expression relation quantifies over the names of resources used for the heaps on the specification and implementation sides. This allows us to allocate fresh instances of ghost resources corresponding to the heaps (for any of the two sides) and simulate the desired part of the program. This is the reason why we can prove such strong equations as Commutativity, Idempotency, Hoisting, etc. The proof of Commutativity above also elucidates the use of deterministic reduction for the specification side.

Theorem 4.4.3 (Idempotency). *If* $\Xi \mid \Gamma \vdash e : \tau$ *then* $\Xi \mid \Gamma \vDash \texttt{let}\, x = e\, \texttt{in}\, (x,x) \approx_{\text{ctx}} (e,e) : \tau \times \tau$

Proof Sketch. We show the contextual equivalence, by proving logical relatedness in both directions. For the left-to-right direction, we allocate a fresh heap and simply simulate twice on the specification side using the same reduction on the implementation side. For the other direction, we simulate the same reduction on the specification side twice for the two different reductions on the implementation side. For the latter we conclude, by determinism of reduction on the specification side, that the two reductions coincide. □

Theorem 4.4.4 (Rec Hoisting). *If* $\Xi \mid \Gamma \vdash e_1 : \tau$ *and* $\Xi \mid \Gamma, y : \tau, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e_2 : \tau_2$ *then*

$$\Xi \mid \Gamma \vDash \texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = e_2 \preceq_{\text{ctx}} \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$$

Proof Sketch. The proof of this theorem is quite tricky, in particular because the the number of operational steps do not match up for the function bodies on the implementation and specification sides. We do not delve into those issues here, but concentrate instead on the high-level structure of the proof.

We prove three different contextual refinements, such that their composition gives us the desired contextual refinement in the theorem. These three contextual refinements are:

(a) $\texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = e_2 \preceq_{\text{ctx}} \texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, z = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$

(b) $\texttt{let}\, y = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, z = e_1\, \texttt{in}\, e_2 \preceq_{\text{ctx}} \texttt{let}\, z = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$

(c) $\texttt{let}\, z = e_1\, \texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 \preceq_{\text{ctx}} \texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in}\, e_2 : \tau_1 \to \tau_2$ where z is a fresh variable.

We prove (a) by proving the corresponding logical relatedness. Since e_1 reduces to a value we know that it will reduce deterministically to some value under any heap on the specification side. We prove (c) also by the corresponding logical relatedness which is rather trivial to prove.

To prove (b) we show the corresponding logical relatedness for a slightly stronger logical relation; $\preceq^{NN}_{\log}$. The NN-logical relation is defined entirely similarly to the primary logical relation above except that the specification side is required to deterministically reduce to a value *in the same number of steps* as the implementation side. Notice that the proofs of the fundamental theorem and soundness for NN-logical relation are very similar to those of the primary logical relation.

Formally, for (b) we show

$$\begin{array}{rl} & \texttt{let}\, y = e_1\, \texttt{in}\texttt{rec}\, f(x) = \texttt{let}\, z = e_1\, \texttt{in} e_2 \\ \preceq^{NN}_{\log} & \texttt{let}\, z = e_1\, \texttt{in}\texttt{rec}\, f(x) = \texttt{let}\, y = e_1\, \texttt{in} e_2 : \tau' \to \tau'' \end{array}$$

This logical relatedness is in fact rather easy to show if we know that all reductions of e_1 (on either side) take the same number of steps. This is precisely why we use the NN-logical relation: By the fundamental theorem of the NN-logical relation we know that $e_1 \preceq^{NN}_{\log} e_1 : \tau' \to \tau''$ and hence we can conclude that both outer reductions (on either side) take the same number of steps, say n. Similarly we know that both reductions of e_1 inside the functions also take the same number of steps, say m. Hence, by allocating appropriate heaps, we can show that the outer reduction of e_1 on the implementation side takes the same number steps as that of the reduction of the inner one on the specification side. This shows, by determinism of reduction on the specification side, that $n = m$, which allows us to conclude the proof. □

Theorem 4.4.5 (η expansion for Rec)**.** *If* $\Xi \mid \Gamma \vdash e : \tau_1 \to \tau_2$ *then* $\Xi \mid \Gamma \vDash e \preceq_{\text{ctx}} \texttt{rec}\, f(x) = e\ x : \tau_1 \to \tau_2$

Proof Sketch. We prove this theorem by proving the following three contextual refinements.

(a) $\Xi \mid \Gamma \vDash e \preceq_{\text{ctx}} \texttt{let}\, y = e\, \texttt{in}\texttt{rec}\, f(x) = (y\ x) : \tau \to \tau'$

(b) $\Xi \mid \Gamma \vDash \texttt{let}\, y = e\, \texttt{in}\texttt{rec}\, f(x) = (y\ x) \preceq_{\text{ctx}} \texttt{rec}\, f(x) = \texttt{let}\, y = e\, \texttt{in}(y\ x) : \tau \to \tau'$

(c) $\Xi \mid \Gamma \vDash \texttt{rec}\, f(x) = \texttt{let}\, y = e\, \texttt{in}(y\ x) \preceq_{\text{ctx}} \texttt{rec}\, f(x) = (e\ x) : \tau \to \tau'$

Refinements (a) and (c) follow rather easily from their corresponding logical relatedness while case (b) is an instance of rec Hoisting above. For (c) notice that f does not appear free in e. □

Theorem 4.4.6 (β reduction for λ). *If* $\Xi \mid \Gamma, x : \tau_1 \vdash e_1 : \tau_2$ *and* $\Xi \mid \Gamma \vdash e_2 : \tau_1$ *then*

$$(\lambda\, x.\, e_1)\ e_2 \ \preceq_{\mathsf{ctx}}\ e_1[e_2/x] \ :\ \tau$$

Proof Sketch. By induction on the typing derivation of e_1; for each case we use appropriate contextual refinements proven by (using the induction hypothesis if necessary) some of the contextual refinement theorems stated above and some instances of logical relatedness. We only present a couple cases here.

Case $e_1 = \mathtt{inj}_i\, e$ The induction hypothesis tells us that $\Xi \mid \Gamma \vDash (\lambda\, x.\, e)\ e_2 \preceq_{\mathsf{ctx}} e[e_2/x] : \tau_i$ and we have to show that $\Xi \mid \Gamma \vDash (\lambda\, x.\, \mathtt{inj}_i\, e)\ e_2 \preceq_{\mathsf{ctx}} (\mathtt{inj}_i\, e)[e_2/x] : \tau_1 + \tau_2$. Notice that it is easy to prove (using the fundamental theorem) that $\Xi \mid \Gamma \vDash (\lambda\, x.\, \mathtt{inj}_i\, e)\ e_2 \preceq_{\mathsf{log}} \mathtt{inj}_i\, ((\lambda\, x.\, e)\ e_2) : \tau_1 + \tau_2$ The final result follows by the induction hypothesis, transitivity of contextual refinement and the fact that contextual refinement is a congruence relation.

Case $e_1 = \mathtt{rec}\, f(y) = e$ The induction hypothesis tells us that $\Xi \mid \Gamma, y : \tau_1, f : \tau_1 \to \tau_2 \vDash (\lambda\, x.\, e)\ e_2 \preceq_{\mathsf{ctx}} e[e_2/x] : \tau_2$ and we have to show that $\Xi \mid \Gamma \vDash (\lambda\, x.\, (\mathtt{rec}\, f(y) = e))\ e_2 \preceq_{\mathsf{ctx}} (\mathtt{rec}\, f(y) = e)[e_2/x] : \tau_1 \to \tau_2$ or equivalently (by simply massaging the terms) $\Xi \mid \Gamma \vDash \mathtt{let}\, x = e_2\, \mathtt{in} (\mathtt{rec}\, f(y) = e) \preceq_{\mathsf{ctx}} (\mathtt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$. By rec Hoisting and transitivity of contextual refinement, it suffices to show $\Xi \mid \Gamma \vDash (\mathtt{rec}\, f(y) = \mathtt{let}\, x = e_2\, \mathtt{in} e) \preceq_{\mathsf{ctx}} (\mathtt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$ which easily follows from the induction hypothesis and the fact that contextual refinement is a congruence relation. □

We omit the theorems of hoisting and η-expansion for polymorphic terms as they are fairly similar in statement and proof to their counterparts for recursive functions. We also omit β-reduction for polymorphic terms and recursive functions. The former follows directly from the corresponding logical relatedness and the latter follows from β-reduction for λ's and rec-unfolding: if $\Xi \mid \Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2$, then

$$\Xi \mid \Gamma \vDash \mathtt{rec}\, f(x) = e \ \preceq_{\mathsf{ctx}}\ \lambda\, x.\, e'[(\mathtt{rec}\, f(x) = e')/f] : \tau_1 \to \tau_2,$$

which is a consequence of the corresponding logical relatedness.

Theorem 4.4.7 (Equations for stateful computations). *See Figure 43.*

Proof. Left identity follows by proving both logical relatednesses. Right identity is proven as follows using equational reasoning:

$$\begin{aligned}
e_2\ e_1 &\preceq_{\mathsf{ctx}} \mathtt{let}\, x = e_2\, \mathtt{in} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}(\mathtt{return}\, y)\, \mathtt{in} \mathtt{let}\, z = x\, y\, \mathtt{in}\, \lambda_.\, z \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, x = e_2\, \mathtt{in} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}(\mathtt{return}\, y)\, \mathtt{in}\, \lambda_.\, \mathtt{let}\, z = x\, y\, \mathtt{in}\, z \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, x = e_2\, \mathtt{in} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}(\mathtt{return}\, y)\, \mathtt{in}\, x \\
&\preceq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{let}\, x = e_2\, \mathtt{in} \mathtt{bind}(\mathtt{return}\, y)\, \mathtt{in}\, x \\
&\preceq_{\mathsf{ctx}} \mathtt{bind}(\mathtt{return}\, e_1)\, \mathtt{in}\, e_2 : \mathsf{ST}\ \rho\ \tau
\end{aligned}$$

Here the second equation is by rec Hoisting and the fourth by a variant of commutativity. The rest follow by proving the corresponding logical relatedness. Associativity is proven as follows using equational reasoning:

$$
\begin{aligned}
&\mathtt{bind}(\mathtt{bind}\, e_1\, \mathtt{in}\, e_2)\, \mathtt{in}\, e_3 \\
&\leq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}\, y\, \mathtt{in} \mathtt{let}\, z = (e_2, e_3)\, \mathtt{in} (\lambda x. \mathtt{bind}(\pi_1\, z)\, x\, \mathtt{in}\, \pi_2\, z) \\
&\leq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}\, y\, \mathtt{in} (\lambda x. \mathtt{let}\, z = (e_2, e_3)\, \mathtt{in} \mathtt{bind}(\pi_1\, z)\, x\, \mathtt{in}\, \pi_2\, z) \\
&\leq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}\, y\, \mathtt{in} (\lambda x. \mathtt{let}\, z_1 = e_2\, \mathtt{in} \mathtt{let}\, z_2 = e_3\, \mathtt{in} \\
&\qquad \mathtt{let}\, z_3 = (z_1\ x) \mathtt{in} \mathtt{bind}\, z_3\, \mathtt{in}\, z_2) \\
&\leq_{\mathsf{ctx}} \mathtt{let}\, y = e_1\, \mathtt{in} \mathtt{bind}\, y\, \mathtt{in} (\lambda x. \mathtt{let}\, z_1 = e_2\, \mathtt{in} \mathtt{let}\, z_3 = (z_1\ x) \mathtt{in} \\
&\qquad \mathtt{let}\, z_2 = e_3\, \mathtt{in} \mathtt{bind}\, z_3\, \mathtt{in}\, z_2) \\
&\leq_{\mathsf{ctx}} \mathtt{bind}\, e_1\, \mathtt{in} (\lambda x. \mathtt{bind}(e_2\ x) \mathtt{in} e_3) : \mathsf{ST}\ \rho\ \tau
\end{aligned}
$$

Here the second equation is by rec Hoisting and the fourth by a variant of commutativity. The rest follow by proving the corresponding logical relatedness. □

4.5 Iris Definitions of Predicates used in the Logical Relation

In this section we detail how the abstract predicates (regions, $\mathsf{region}(r, \gamma_h, \gamma_h')$, $\mathsf{isRgn}(\alpha, r)$, $heap_{\gamma_h}(h)$ and $\ell \mapsto_\gamma v$) used in the definition of the logical relation are precisely defined in the Iris logic. To this end, we first introduce three more concepts from the Iris logic: invariants, saved predicates and ghost-state.

Invariants, Saved Predicates and Ghost State

We extend the grammar for Iris propositions P, presented in §4.3 with syntax for invariants, saved predicates and ghost-resources:

$$P ::= \cdots \mid \boxed{P} \mid \gamma \rightarrowtriangle \Phi \mid \checkmark(a) \mid \boxed{a : \mathcal{M}}^{\gamma}$$

Invariants in Iris, $\boxed{P}$, are typically used to enforce that a proposition P holds for some shared state. In this paper we use a certain kind of invariants for which we can use the following rules for allocating and opening invariants[7]:

$$\textsc{Inv-alloc}\ \frac{P}{\Rightarrow\!\!\!\triangleright \boxed{P}} \qquad \textsc{Inv-open}\ \frac{P \Rrightarrow P * Q}{\boxed{P} \Rrightarrow Q}$$

[7] Technically, $\Rightarrow$ has masks $\Rightarrow_{\mathcal{E}}$ where $\mathcal{E}$ keeps track of already opened invariants, preventing the same invariant being opened twice in a nested fashion, which would be unsound. In this paper we omit the masks for the sake of simplicity.

Notice that these are not the general rules for allocating and opening invariants in Iris. In general, the rule Inv-open should involve a $\triangleright$ to ensure soundness of the logic. However, the above rules do hold for the invariants we use in this paper.[8] Invariants are persistent, $\boxed{P} \dashv\vdash \boxed{P} * \boxed{P}$.

For storing of Iris propositions we use a mechanism called saved predicates, $\gamma \Mapsto \Phi$. This is simply a convenient way of assigning a name γ to a predicate Φ. There are only three rules governing the use of saved propositions. We can allocate them (rule SavedPred-Alloc), they are persistent (rule SavedPred-Persistent) and the association of names to predicates is functional (rule SavedPred-Equiv).

SavedPred-Alloc
$\Rrightarrow_{\mathcal{E}} \exists \gamma.\ \gamma \Mapsto \Phi$

SavedPred-Persistent
$\gamma \Mapsto \Phi \dashv\vdash \gamma \Mapsto \Phi * \gamma \Mapsto \Phi$

SavedPred-Equiv
$$\frac{\gamma \Mapsto \Phi * \gamma \Mapsto \Psi}{\triangleright \Phi(a) \vdash \triangleright \Psi(a)}$$

The later modality is used in rule SavedPred-Equiv as a guard to avoid self referential paradoxes [38], which is not so surprising, after all, since saved propositions essentially allow us to store a predicate (something of type $\kappa \to iProp$) inside a proposition (something of type $iProp$).

Resources in Iris are described using a kind of partial commutative monoids, and the user of the logic can introduce new monoids. For instance, in the case of finite partial maps, the partiality comes from the fact that disjoint union of finite maps is partial. Undefinedness is treated by means of a validity predicate $\checkmark : \mathcal{M} \to iProp$, which expresses which elements of the monoid $\mathcal{M}$ are valid/defined.

We write $\boxed{a : \mathcal{M}}^{\gamma}$ to assert that a monoid instance named γ, of type $\mathcal{M}$ has contents a. Often, we disregard the type if it is obvious from the context. We think of this assertion as a ghost variable γ with contents a.

Ghost-Alloc
$\checkmark a \vdash \Rrightarrow \exists \gamma.\ \boxed{a}^{\gamma}$

Own-Valid
$\boxed{a}^{\gamma} \vdash \checkmark(a)$

Sharing
$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$

Some Useful Monoids In this paragraph, we describe a few monoids which are particularly useful and which we will use in the following. We do not give the full definitions of the monoids (those can be found in [38]), but focus instead on the properties which the elements of the monoids satisfy, shown in Figure 47. These rules stated are only for monoids that we use in this work and not in Iris in its generality. For instance, in the rule Auth-Included, $\subseteq$ is a set relation and is defined for finite set and finite partial function monoids and not in general.

The figure depicts the rules necessary for allocating and updating finite set monoids, $\mathsf{finset}(A)$, and finite partial function monoids, $A \rightharpoonup^{\mathsf{fin}} M$. In

[8] The rules hold for invariants $\boxed{P}$ where P is *timeless*. For details see [38].

AUTH-INCLUDED
$\bullet a \cdot \circ b \vdash b \subseteq a$

FPFN-VALID
$\checkmark(a) \dashv\vdash \forall x \in \mathrm{dom}(a).\ \checkmark(a(x))$

AGREEMENT-VALID
$\checkmark(\mathrm{ag}(a) \cdot \mathrm{ag}(b)) \dashv\vdash a = b$

EXCLUSIVE
$\not\checkmark(\mathsf{ex}(a) \cdot b)$

FRAG-DISTRIBUTES
$\circ a \cdot \circ b = \circ(a \cdot b)$

FULL-EXCLUSIVE
$\not\checkmark(\bullet a \cdot \bullet b)$

AUTH-ALLOC-FINSET
$$\frac{h \cap a = \emptyset}{\boxed{\bullet h}^{\gamma} \Rrightarrow \boxed{\bullet (h \uplus a) \cdot \circ a}^{\gamma}}$$

AUTH-ALLOC-FPFN
$$\frac{\mathrm{dom}(h) \cap \mathrm{dom}(a) = \emptyset}{\boxed{\bullet h}^{\gamma} \Rrightarrow \boxed{\bullet (h \uplus a) \cdot \circ a}^{\gamma}}$$

AGREE
$\mathrm{ag}(a) \cdot \mathrm{ag}(a) = \mathrm{ag}(a)$

FPFN-OPERATION-SUCCESS
$$(a \cdot b)(x) = \begin{cases} a(x) & \text{if } x \in \mathrm{dom}(a) \wedge x \notin \mathrm{dom}(b) \\ a(x) \cdot b(x) & \text{if } x \in \mathrm{dom}(a) \cap \mathrm{dom}(b) \\ b(x) & \text{if } x \in \mathrm{dom}(b) \wedge x \notin \mathrm{dom}(a) \end{cases}$$

AUTH-UPDATE-FPFN
$$\boxed{\bullet (h \uplus (\ell \mapsto \mathsf{ex}(v_1))) \cdot \circ \ell \mapsto \mathsf{ex}(v_1)}^{\gamma} \Rrightarrow \boxed{\bullet (h \uplus (\ell \mapsto \mathsf{ex}(v_2))) \cdot \circ \ell \mapsto \mathsf{ex}(v_2)}^{\gamma}$$

Figure 47: Rules for selected monoid resources in Iris

these monoids, the monoid operation $x \cdot y$ is *disjoint* union. The notation $a \mapsto b : A \rightharpoonup^{\mathrm{fin}} B \triangleq \{(a, b)\}$ is a singleton finite partial function.

The constructs $\bullet$ and $\circ$ are constructors of the so-called authoritative monoid $\textsc{Auth}(M)$. We read $\bullet a$ as *full* a and $\circ a$ as *fragment* a. We use the authoritative monoid to distribute ownership of fragments of a resource. The intuition is that $\bullet a$ is the authoritative knowledge of the full resource, think of it as being kept track of in a central location. This central location is the full part of the resource (see rule AUTH-INCLUDED). The fragments, $\circ a$, can be shared (rule FRAG-DISTRIBUTES) while the full part (the central location) should always remain unique (rule FULL-EXCLUSIVE).

In addition to authoritative monoids, we also use the agreement monoid $\textsc{Ag}(M)$ and exclusive monoid $\mathrm{Ex}(M)$. As the name suggests, the operation of the agreement monoid guarantees that $\mathrm{ag}(a) \cdot \mathrm{ag}(b)$ is invalid whenever $a \neq b$ (and otherwise it is idempotent; see rules AGREE and AGREEMENT-VALID). From the rule AGREE it follows that the ownership of elements of $\textsc{Ag}(M)$ is persistent.

$$\boxed{\mathrm{ag}(a)}^{\gamma} \dashv\vdash \boxed{\mathrm{ag}(a) \cdot \mathrm{ag}(a)}^{\gamma} \dashv\vdash \boxed{\mathrm{ag}(a)}^{\gamma} * \boxed{\mathrm{ag}(a)}^{\gamma}$$

The operation of the exclusive monoid never results in a valid element (rule EXCLUSIVE), enforcing that there can only be one instance of it owned. We can now give meaning to the heap-specific predicates used in the earlier sections,

by presenting the canonical example of a Heap monoid:

$$\textsc{Heap} \triangleq \textsc{Auth}(Loc \xrightarrow{\text{fin}} (\textsc{Ex}(Val)))$$
$$heap_\gamma(h) \triangleq \boxed{\bullet h}^\gamma \qquad \ell \mapsto_\gamma v \triangleq \boxed{\circ [\ell \mapsto \mathsf{ex}(v)]}^\gamma$$

Notice here that Heap is build from nesting Ex in the finite partial functions monoid, which again is nested in the Auth monoid. Therefore, to allocate and update and in the Heap monoid, we can use Auth-Alloc-Fpfn and Auth-Update-Fpfn respectively.

Encoding of Regions by Ghost Resources

In order to concretely represent bijections and relatedness between locations, we use a pair of monoids, one for the bijection (one-to-one correspondence) and one for the semantic interpretation, i.e., a name to a saved predicate:

$$\mathsf{Rel} \triangleq \textsc{Auth}((Loc \times Loc) \xrightarrow{\text{fin}} (\textsc{Ag}(Names))) \qquad \mathsf{Bij} \triangleq \textsc{Auth}(\mathcal{P}(Loc \times Loc))$$

Both are defined as authorative monoids which allow for having a global and a local part. To tie the two monoids together with a semantic region r (the name r is simply a positive integer) we use a third monoid:

$$\mathsf{Region} \triangleq \textsc{Auth}(\mathbb{Z}^+ \xrightarrow{\text{fin}} (\textsc{Ag}(Names \times Names)))$$

We fix a global ghost name γ_{reg} for an instance of this last monoid. For Region, ownership of $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ indicates that the semantic region r is represented by two ghost variables named γ_{bij} and γ_{rel}, for Bij and Rel respectively. Notice that this ownership of $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ is duplicable and also, due to the properties of the agreement monoid, we have that the semantic region tied to r is uniquely defined. Formally,

$$\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma'_{bij}, \gamma'_{rel})}^{\gamma_{reg}} \vdash \gamma_{bij} = \gamma'_{bij} \wedge \gamma_{rel} = \gamma'_{rel} \tag{4.6}$$

We can now present the $\mathsf{region}(r, \gamma_h, \gamma'_h)$ predicate in detail:

$$\mathsf{region}(r, \gamma_h, \gamma'_h) \triangleq \exists R, \gamma_{bij}, \gamma_{rel}.\ \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel}) : \mathsf{Region}}^{\gamma_{reg}} * \boxed{\bullet\, R : \mathsf{Rel}}^{\gamma_{rel}} * \left[*\right]_{(\ell,\ell') \mapsto \mathsf{ag}(\gamma_{pred}) \in R} \Big(\exists \Phi : (Val \times Val) \to iProp), v, v'.\ \ell \mapsto_{\gamma_h} v * \ell' \mapsto_{\gamma'_h} v' * \gamma_{pred} \Rrightarrow \Phi * \triangleright\, \Phi(v, v') \Big)$$

The predicate asserts that the semantic region r is associated with two ghost names, γ_{bij} and γ_{rel}, by $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, and full authoritative ownership of R, which is a mapping of pairs of locations to ghost names. Further, for each element $(\ell, \ell') \mapsto \mathsf{ag}(\gamma_{pred}) \in R$ we have ownership of the points-to predicates

$\ell \mapsto_{\gamma_h} v$ and $\ell' \mapsto_{\gamma'_h} v'$ and the knowledge about a saved predicate Φ, named by γ_{pred}, that holds later for v and v'.

The regions predicate keeps track of all the allocated regions by having the full authoritative part $\boxed{\bullet M : \mathsf{Reg}}^{\gamma_{reg}}$:

$$\mathsf{regions} \triangleq$$

$$\boxed{\exists M.\ \boxed{\bullet M : \mathsf{Region}}^{\gamma_{reg}} * \mathop{\Large *}_{r \mapsto \mathsf{ag}(\gamma_{bij},\gamma_{rel}) \in M} \left(\begin{array}{l} \exists g : \mathsf{finset}(Loc \times Loc), R : (Loc \times Loc) \xrightarrow{\text{fin}} (\textsc{Ag}(Names)).\\ \boxed{\bullet g}^{\gamma_{bij}} * \mathsf{bijection}(g) * \boxed{\circ R}^{\gamma_{rel}} * g = \mathrm{dom}(R) \end{array} \right)}$$

For each element $r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})$ in M, regions have full authoritative ownership of a bijection g and fragment ownership of R, which maps each pairs of locations to a ghost name for saved predicates. Here, g and the domain of R is forced to be equal, ensuring that all pairs that are related in the bijection are also related in the region. Notice that since the regions predicate is an invariant, it is also persistent.

Notice here as well that individual regions are tied to the regions predicate, regions, by having the fragment ownership of $\boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel}) : \mathsf{Region}}^{\gamma_{reg}}$ since the authoritative element $\boxed{\bullet M : \mathsf{Region}}^{\gamma_{reg}}$ is owned by regions. Similarly, the regions predicate is tied to all regions by asserting ownership of the fragment $\boxed{\circ R}^{\gamma_{rel}}$. This illustrates how ghost resources are important to enforce relations in and out of invariants.

We can now give meaning to the abstract predicates used in the definition of $\mathtt{STRef}\ \rho\ \tau$[9]:

$$\begin{aligned}
\mathsf{isRgn}(\alpha, r) &\triangleq \exists \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ \alpha = r * \boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}\\
\mathsf{bij}(r, \ell, \ell') &\triangleq \exists \gamma_{bij}, \gamma_{rel}.\ \boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ (\ell, \ell')}^{\gamma_{bij}}\\
\mathsf{rel}(r, \ell, \ell', \Phi) &\triangleq \exists \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ \boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} *\\
&\qquad \boxed{\circ [(\ell, \ell') \mapsto \mathsf{ag}(\gamma_{pred})]}^{\gamma_{bij}} * \gamma_{pred} \Rrightarrow \Phi
\end{aligned}$$

Each of the predicates owns the ghost resource suggested by its name. For instance, Property (4.5) from §4.3 can now be shown:

$$\mathsf{regions} \Rrightarrow \exists r.\ \mathsf{region}(r, \gamma_h, \gamma'_h)$$

First, we open the invariant using Inv-open to obtain $\boxed{\bullet M : \mathsf{Region}}^{\gamma_{reg}}$. By Ghost-Alloc we obtain $\boxed{\bullet \emptyset \cdot \circ \emptyset : \mathsf{Rel}}^{\gamma_{rel}}$ and $\boxed{\bullet g}^{\gamma_{bij}}$, for fresh ghost names γ_{rel} and γ_{bij}. Now, by Auth-Alloc-Fpfn we can extend M with $r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})$,

[9] The predicate $\boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ appears in all the abstract predicates to obtain γ_{bij} and γ_{rel}. This is to keep the initial description of the predicates simple. The redundancy does not exist in the actual implementation.

to obtain $\boxed{\circ r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, for some r not in $\mathrm{dom}(M)$, since M is finite. $\mathsf{region}(r, \gamma_h, \gamma_h')$ now holds trivially, since there are no locations allocated in $\boxed{\bullet \emptyset}^{\gamma_{rel}}$. Similarly, $\mathsf{bijection}(\emptyset)$ and $\mathrm{dom}(\emptyset) = \emptyset$ hold trivially, so we have reestablished the body of the invariant.

4.6 Formalization in Coq

We have formalized our technical development and proofs in the Iris implementation in Coq [38, 39]. The Iris implementation in Coq [38] includes a model of Iris and proof of soundness of the Iris logic itself. The Iris Proof Mode (IPM) [39] allows users to carry out proofs inside Iris in much the same way as in Coq itself by providing facilities for working with the substructural contexts and modalities of Iris. We have used Iris and IPM to formalize the future modality, the IC predicates, our logical relation and to prove the state-independence theorem and all the refinements presented in this paper.

The Trusted Computing Base

Even though our logical relation has been defined inside the Iris logic, the soundness theorem of Iris [38] allows us to prove the soundness of our logical relation:

```
Theorem binary_soundness Γ e e' τ : typed Γ e τ → typed Γ e' τ →
  (∀ Σ `{ICG_ST Σ, LogRelG Σ}, Γ ⊨ e ≤log≤ e' : τ) → Γ ⊨ e ≤ctx≤ e' : τ.
```

This statement says that whenever $\Xi \mid \Gamma \vdash e : \tau$ and $\Xi \mid \Gamma \vdash e' : \tau$ and we can prove in the Iris logic (notice the quantification of Iris parameters, `Σ `{ICG_ST Σ} `{LogRelG Σ}`)[10] that e and e' are logically related, then e contextually refines e'. Notice that Ξ does not appear in the Coq code as we are using de Bruijn indices to represent type variables and hence need no type level context. The definition of contextual refinement and well-typedness are in turn normal Coq statements, independent of Iris.

All lemmas and theorems in this paper are type checked by Coq without any assumptions or axioms apart from the use of functional extensionality which is used for the de Bruijn indices. It is used by the Autosubst library.

Extending Iris and IPM and instantiating them with STLang

The implementations of Iris and IPM in Coq are almost entirely independent of the choice of programming language. In practice, the only definitions that are parameterized by a language are the definitions of weakest-precondition and Hoare triples. To use these with a particular programming language, one needs to instantiate a data structure in Coq that represents the language.

[10] Σ is the set of Iris resources and the other two parameters express that resources necessary for IC and our logical relations are present in Σ.

Basically, one is required to instantiate this data structure with the language's set of states (heaps in our case), expressions, values and reduction relation, together with proofs that they behave as expected (e.g., values do not reduce any further). In this work we use IC predicates and IC triples instead of the weakest precondition and Hoare triples used in earlier work. Therefore, we have also parameterized IC predicates and IC triples by a data structure representing the programming language. We instantiate these with STLang.

The formalization of Iris in Coq is a shallow embedding. That is, the model of the Iris logic is formalized in Coq, and terms of the type *iProp* (propositions of Iris) are defined as well-behaved predicates over the elements of that model. The advantage of shallow embeddings is that one can easily introduce new connectives and modalities to the logic by defining another function with *iProp* as co-domain. For instance, our IC predicate is defined as follows in Coq.

```
Definition ic_def {Λ Σ} '{ICState Λ, ICG Λ Σ} γ E e Φ : iProp Σ :=
(∀ σ1 σ2 v n, (⌜nsteps pstep n (e, σ1) (of_val v, σ2)⌝ ∗ ownP_full γ σ1)
      -∗ |={E}=[n]=> Φ v n ∗ ownP_full γ σ2)%I.
```

Here, $\ulcorner \cdot \urcorner$ embeds Coq propositions into Iris and `ownP_full` γ σ is the full ownership of the physical state of the language (parameter Λ), equivalent to our $heap_\gamma(\sigma)$. The Coq proposition `nsteps pstep n (e,` $\sigma 1$`) (of_val v,` $\sigma 2$`)` states that (e, σ_1) physically reduces (`pstep`) in n steps to (v, σ_2) where v is a value. The `%I` at the end instructs Coq to parse connectives (e.g., the universal quantification) as Iris connectives and not those of Coq.

As discussed in [39], IPM tactics, like the `iMod` tactic for elimination of modalities, simply apply lemmas with side conditions that are discharged with the help of Coq's type class inference mechanism. Extending IPM with support for the future modality and IC predicates essentially boils down to instantiating some of these type classes appropriately.

Representing binders

We use de Bruijn indices to represent variables both at the term level and the type level; in particular, we use the Autosubst library [68]. It provides excellent support for manipulating and simplifying terms with de Bruijn indices in Coq. The simplification procedure, however, seems to be non-linear in the size of the term. This is the main reason for the slowness of Coq's processing of our proofs.[11]

[11] About 17 minutes on a laptop using "make -j4" to compile our Coq formalization of about 12,500 lines.

4.7 Related work

The most closely related work is the original seminal work of Launchbury and Jones [47], which we discussed and related to in the Introduction. In this section we discuss other related work.

Moggi and Sabry [52] showed type soundness of calculi with `runST`-like constructs, both for a call-by-value language (as we consider here) and for a lazy language. The type soundness results were shown with respect to operational semantics in which memory is divided into regions: a `runST`-encapsulated computation always start out in an empty heap and the final heap of such a computation is thrown away. Thus their type soundness result does capture some aspects of encapsulation. However, the models in *loc. cit.* are not relational and therefore not suitable for proving relational statements such as our theorems above. The authors write: *"Indeed substantially more work is needed to establish soundness of equational reasoning with respect to our dynamic semantics (even for something as unsurprising as β-equivalence)"* [52].

In contrast to Moggi and Sabry [52], who also considered type soundness for a call-by-need language, we only develop our model for a call-by-value language. For call-by-need one would need to keep track of the dependencies between effectful operations in the operational semantics and *only* evaluate them if they contribute to the end result. These dependencies would also have to be reflected in the logical relations model. It is not clear how difficult that would be and we believe it deserves further investigation.

It was pointed out already in [47] that there seems to be a connection between encapsulation using `runST` and effect masking in type-and-effect systems à la Gifford and Lucassen [26]. This connection was formalized by Semmelroth and Sabry [69], who showed how a language with a simplified type-and-effect system with effect masking can be translated into a language with `runST`. Moreover, they showed type soundness on their language with `runST` with respect to an operational semantics. In contrast to our work, they did not investigate relational properties such as contextual refinement or equivalence.

Benton *et al.* have investigated contextual refinement and equivalence for type-and-effect systems in a series of papers [10–13] and their work was extended by Thamsborg and Birkedal [74] to a language with higher-order store, dynamic allocation and effect masking. These papers considered soundness of some of the contextual refinements and equivalences for pure computations that we have also considered in this paper, but, of course, with very different assumptions, since the type systems in *loc. cit.* were type-and-effect systems. Thus, as an alternative to the approach taken in this paper, one could also imagine trying to prove contextual equivalences in the presence of `runST` by translating the type system into the language with type-and-effects used in [74] and then appeal to the equivalences proved there. We doubt, however, that such an alternative approach would be easier or better in any way. The

logical relation that we define in this paper uses an abstraction of regions and relates regions to the concrete global heap used in the operational semantics. At a very high level, this is similar to the way regions are used as an abstraction in the models for type-and-effect systems, e.g., in [74]. However, since the models are for different type systems, they are, of course, very different in detail. One notable advance of the current work over the models for type-and-effect systems, e.g., the concrete step-indexed model used in [74], is that our use of Iris allows us to give more abstract proofs of the fundamental lemma for contextual refinements than a more low-level concrete step-indexed model would.

Recently Iris has been used in other works to define logical relations for different type systems than the one we consider here [39, 42]. The definitions of logical relations in those works have used Iris's weakest preconditions $\mathsf{wp}\, e \{v.\, P\}$ to reason about computations. Here, instead, we use our if-convergence predicate, $\mathsf{IC}^{\gamma}\, e \{v.\, P\}$. One of the key technical differences between the weakest precondition predicate and the if convergence predicate is that the latter keeps explicit track of the ghost variable γ used for heap. This allows us to reason about different (hypothetical) runs of the same expression, a property we exploit in the proofs of contextual refinements in §4.4.

4.8 Conclusion and Future Work

We have presented a logical relations model of STLang, a higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with `runST`. To the best of our knowledge, this is the first model which can be used to show that `runST` provides proper encapsulation of state, in the sense that a number of contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of stateful computations encapsulated using `runST`. We defined our logical relation in Iris, a state-of-the-art program logic. This greatly simplified the construction of the logical relation, e.g., because we could use Iris's features to deal with the well-known type-world circularity. Moreover, it provided us with a powerful logic to reason in the model. Our logical relation and our proofs of contextual refinements used several new technical ideas: in the logical relation, e.g., the linking of the region abstraction to concrete heaps and the use of determinacy of evaluation on the specification side; and, in the proof of contextual refinements, e.g., the use of a helper-logical relation for reasoning about equivalence of programs using the same number of steps on the implementation side and the specification side. Finally, we have used and extended the Iris implementation in Coq to formalize our technical development and proofs in Coq.

Future work Future work includes developing a model for a call-by-need variant of STLang. In the original paper [47], Launchbury and Peyton Jones argue that it would be useful to have a combinator for parallel composition of stateful programs, as opposed to the sequential composition provided by the monadic bind combinator. One possible direction for future work is to investigate the addition of concurrency primitives in the presence of encapsulation of state. It is not immediately clear what the necessary adaptations are for keeping the functional language pure. It would be interesting to investigate whether a variation of the parallelization theorem studied for type-and-effect systems in [42] would hold for such a language.

Chapter 5

Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems

MORTEN KROGH-JESPERSEN, Aarhus University, Denmark
AMIN TIMANY, imec-Distrinet, KU Leuven, Belgium
MARIT EDNA OHLENBUSCH, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

The formal development accompanying this research project can be found at https://github.com/mkroghj/aneris.

Abstract

Building network-connected programs and distributed systems is a powerful way to provide availability in our digital, always-connected era. As always, however, with great power comes great complexity. As such, reasoning about distributed systems is well-known to be quite difficult.

In this paper we present Aneris, a state-of-the-art separation logic capable of *node-local reasoning* about concurrent and distributed systems. The logic is higher-order, concurrent, with higher-order store, and network sockets and is built entirely in the Coq proof-assistant.

We use our logic to verify a load-balancer that use threading to distribute load amongst servers and to verify an implementation of the *two-phase-commit* protocol with a verified replicated logging service as client.

The two examples certifies that Aneris is well-suited for both horizontal and vertical modular reasoning.

5.1 Introduction

Network-connected applications and in particular distributed systems are used every day by myriads of people for providing financial services, hailing taxis and sending messages over social media. However, formal reasoning about such systems is well-known to be difficult because of the complexity that concurrent, stateful programs expose. A well-known approach to combat complexity in programs is to disentangle software systems into independent modules to enable local reasoning. Local reasoning enables (a) information hiding and abstraction and (b) separation of concerns.

Previous work on verification of distributed systems has traditionally focused on verification of protocols of core network components. This approach has proven particularly useful within the context of model-checking, by validating both safety and liveness assertions [59], such as SPIN Holzmann [30], TLA+ [46] and Mace [37]. More recently, significant contributions has been made in the field of formal proofs of implementations of challenging protocols, such as two-phase-commit, lease-based key-value stores, Paxos and Raft [28, 50, 62, 70, 81]. All of these developments define domain specific languages, specialized for distributed systems verification. Protocols and modules proven correct can be compiled to an executable often relying on some trusted code-base. However, reusing verified components in larger applications is difficult because: (1) the implementation specific code is often mixed with logical assertions about the abstract state, therefore lacking abstraction (a), (2) the DSL's do not support many modern language constructs such as concurrency or simplistic local state, (3) combining artifacts requires a unified framework to reason about specifications written in separate DSL's. The Disel framework [70] is a promising candidate for modular programming of distributed components, however, node-local references are visible in the global protocol specifications and the language lacks node-local concurrency.

In this work, we present Aneris, a framework for verifying real-world network-connected applications in Iris [38], specifically developed with modular reasoning in mind. We obtain this property by what we refer to as **node-local reasoning**, similar to *thread-local reasoning* for thread-concurrent programs. We start by motivating the need for Aneris.

Why Aneris?

The design goal of Aneris and the surface language AnerisLang is to facilitate verified modular programming of large software systems, including distributed systems. Thus, Aneris should strive for the following:

1. *The programming language should be realistic, familiar and easy to work with in practice*: In particular, AnerisLang should have higher-order

functions, local state, concurrency and network primitives, to aid the developer in writing succinct, performant programs.

2. *Code re-use and vertical composition of specifications*: A client should be able to use a verified component by only relying on the components specification, known as *modular reasoning* or vertical composition. This allows for changing and updating modules, without having to update the client or the clients proof.

3. *Horizontal composition*: A verified component should be able to be composed with other components, potentially engaging in different protocols, as long as the environment satisfies the protocols stated by the component. This allows for components to be composed horizontally and to build an verify large scale distributed systems.

4. *Verification should be as easy as possible*: Interactive theorem proving, such as tactics in Coq, has proven successful for large scale verification projects. Ideally, using Aneris for verification purposes should provide a compatible experience.

There are many different ways of adding network primitives to languages. One approach is *message-passing*, either by first-class communication channels from the π-calculus or by an implementation of the actor model, similar to Erlang. However, any such implementation is an abstraction built on top of network sockets. Network sockets are a quintessential part of distributed systems and all major operating systems provide an application programming interface (API) for it. AnerisLang provides support for network sockets by exposing a simple API with the core methods necessary for UDP based programming. This allows for a wide-range of real-world systems and protocols to be written (and verified) in AnerisLang.

Aneris is built on top of Iris, a state-of-the-art higher-order concurrent separation logic, which already has powerful built-in features to support reasoning about higher-order programs with higher-order store and concurrency, *e.g.*, higher-order impredicative invariants, higher-order Hoare triples, *etc.*. Setting up such a powerful program logic is difficult in general because it requires one to solve recursive domain equations – however, this has been solved in Iris. Thus, we can obtain all the features mentioned in bullet point (1) above by setting up Aneris on top of Iris and including an API for sockets. We discuss in Section 5.4 how Iris's facilities are used to define the program logic of Aneris.

The higher-order concurrent distributed separation logic of Aneris provides a clear distinction between the programs, specifications for those programs and proofs thereof. Hoare-style reasoning, a traditional way of giving abstract specifications to implementations, can be encoded in Aneris through

weakest-pre-conditions, allowing one to compose proofs about programs vertically without relying on a specific components' implementation, satisfying (2).

Formal reasoning about nodes in distributed systems has often been done by giving an abstract model in the form of some kind of *state-transition system* or *flow-chart*, in the tradition of Floyd [25], Lamport [44, 45]. States are normally taken to be a view of the global state and events are then observable changes to this state. State-transition systems are quite versatile and have been used in other verification applications, *e.g.*, logical relations models [3, 22] and in Hoare-style logic and type-theory [70, 72]. However, Jung et al. [33] showed that all you need is monoids (to encode resources) and invariants[1] (to encode protocols with the help of resources). We follow said approach, and associate each socket with a protocol in the form of an Aneris predicate on the incoming messages. We further allow the network to own the resources in transit, thus resources are transferred from the sender to the network before they are transferred to the recipient. This allows for independent program verification (state evolution) and local synchronization by ownership transfer. This enables reasoning about distributed systems in a fully node-local way (3).

Finally, by the virtue of using Iris as a basis for Aneris we have the Iris proof mode (IPM) [39] at our disposal. It enables us to carry out interactive reasoning about the distributed separation logic of Aneris. This makes verification of distributed systems more pleasant and intuitive (4).

In summary, the key contributions of this work are:

- AnerisLang, a formalized higher-order functional programming language, with higher-order store, concurrency and network sockets, allowing for dynamic creation and binding of sockets to addresses with serialization and de-serialization primitives for encoding and parsing messages.
- Aneris, the first higher-order, concurrent, separation logic with support for network sockets, verified fully in the proof-assistant Coq. Since the logic is built on top of Iris, assertions on state and protocols can use all of the features from Iris, including invariants and monoids.
- A simple, novel, approach to guarding network sockets as predicates on messages, allowing for logical synchronization by the means of ownership-transfer. Ultimately, this enables what we refer to as **node-local reasoning**, the basic principle that allows for *modular reasoning* of distributed systems components.

[1] This has since been reduced to just resources as invariants can be, and are in recent version of Iris, implemented using high-order ghost resources [34, 38].

- We use Aneris to verify a load-balancer, a program that distributes work on multiple servers by the means of threaded concurrency. The only assumption made on the servers is that they have a known address and that the socket protocols do not assert anything about the sender.
- We use Aneris to prove an implementation of *two-phase-commit* correct. Specifically, we prove that the coordinator and participant components satisfy the protocol. We then use these components in a distributed client of the two-phase-commit that does replicated logging, showing that vertical composition is achieved through node-local reasoning.

The structure of the rest of the paper We start be describing the core concepts of Aneris in 5.2. We then show the operational semantics of AnerisLang 5.3 before showing adequacy and how to encode Aneris in Iris in 5.4. We then use the logic to show a specification for a load-balancer 5.5 and two-phase-commit 5.6 with a client of replicated logging 5.7 before describing related work 5.8 and concluding 5.9.

5.2 The core concepts of Aneris

In this section, we present Aneris' approach to formal verification of distributed systems: node-local reasoning and protocols. These concepts are already familiar in the context of separation logic, thus we start by describing local reasoning in this context. We then explain how to lift thread-local reasoning to *node-local* reasoning, a novel approach to distributed systems verification. Finally, we describe protocols in Aneris and show a concrete lock server with a guarding protocol.

Local Reasoning and Thread Local Reasoning

Arguably the most important feature of (concurrent) separation logic, to which it owes its success and prevalence, is that it enables modular reasoning. Originally, separation logic [66] was introduced to enable modular reasoning about the heap. The essential idea was that we could give a local specification $\{P\}\, e\, \{x.\ Q\}$ to a program e involving only the *footprint* of e. Local specifications could then be lifted to (more) global specifications by the following Frame-rule:

$$\textsc{Frame-rule}\quad \frac{\{P\}\, e\, \{x.\ Q\}}{\{P * R\}\, e\, \{x.\ Q * R\}}$$

Here, the proposition R is called *the frame*. The symbol $*$ is separating conjunction. Intuitively, $P * Q$ holds if resources (in this case heaps) can be divided into two disjoint resources such that P holds for one and Q holds for the other.

Thus, the Frame-rule essentially says that executing e for which we know $\{P\}\, e\, \{x.\ Q\}$ cannot possibly affect parts of the heap that are *separate* from its footprint.

Ever since its introduction, separation logic has been extended to resources beyond the heap of the program. Concurrent separation logics [56] have been developed to reason about concurrent programs and again a pre-eminent feature of these program logics is modular reasoning, in this instance with respect to concurrency, *i.e.*, *thread-local* reasoning. That is, when reasoning about a concurrent program we consider threads one at a time and need not reason about different interleavings explicitly. In a way, our frame here is, in addition to the shared fragments of heap and other resources, the execution of other threads which can be interleaved throughout the execution of the thread being verified. This can be seen in the following Fork-rule:

$$\textsc{Fork-rule}\quad \frac{\{P\}\, e\, \{x.\ \mathsf{true}\}}{\{P\}\, \mathsf{fork}\ \{e\}\ \{x.\ x = ()\}}$$

One notable program logic in the family of concurrent separation logics is Iris. Iris is a concurrent higher-order separation logic framework which was designed to reason about concurrent higher-order imperative programming languages. Iris has already proven to be quite versatile for reasoning about a number of sophisticated properties of programming languages, *e.g.*, [35, 36, 75]. In order to support modular reasoning about concurrent programs, Iris (1) features *impredicative invariants* for expressing protocols among multiple threads, and, (2) allows encoding of *higher-order ghost state* using a form of partial commutative monoids for reasoning about resources. We will give examples of these features and explain them in more detail later on.

Aneris programs are higher-order imperative concurrent programs that run on multiple nodes in a distributed system. One of the main contributions of the present work is that when reasoning about distributed systems in Aneris, alongside *heap-local* and *thread-local* reasoning we reason *node-locally*. That is, when proving correctness of Aneris programs we reason about each node of the system in isolation.

In the rest of this section we explain at a conceptual level, how we achieve node-local reasoning in Aneris. The key idea is that although distributed systems and concurrent programs are vastly different, they conceptually share some essential features. The similarities allow us to put the machinery that Iris provides for thread-reasoning about concurrent programs into use for node-local reasoning about distributed systems.

Node-Local Reasoning About Distributed Systems

By the virtue of working in Iris the reasoning in Aneris is both modular with respect to separation logic frames and with respect to threads. Similarly to

threads Aneris allows for *node-local* reasoning about programs:

$$\textsc{Start-rule}\ \frac{\{P * \mathit{freePorts}(ip, \{p | 0 \le p \le 65536\})\}\,\langle n; e\rangle\,\{x.\ \mathsf{true}\}}{\{P * \mathit{freeIp}(ip)\}\,\langle \mathfrak{S}; \mathsf{start}\ \{n; ip; e\}\rangle\,\{x.\ x = \langle \mathfrak{S}; ()\rangle\}}$$

Here start is the command that launches a new node named n in the distributed system associated with ip-address ip running program e. Only the distinguished system node $\mathfrak{S}$ can start new nodes. The idea is that in Aneris, the execution of the system starts with the execution of $\mathfrak{S}$ as the only node in the distributed system. $\mathfrak{S}$ then bootstraps the distributed system by starting other nodes. In order to start a new node associated with ip-address ip, one needs to provide $\mathit{freeIp}(ip)$ which indicates that the ip-address ip is not used by other nodes. The node can on the other hand rely on the fact that when it starts, all ports on the ip-address ip are available. To facilitate modular reasoning, free ports can be divided up:

$$\frac{A \cap B = \emptyset}{\mathit{freePorts}(ip, A) * \mathit{freePorts}(ip, B) \dashv\vdash \mathit{freePorts}(ip, A \cup B)}$$

where $\dashv\vdash$ is logical equivalence of Iris propositions.

In Aneris we associate with each socket (pair of ip-addresses and ports) a protocol which restricts what can be communicated over that socket. In Aneris terms, we write $s \Mapsto^{\mathsf{prot}} \Phi$ to mean that socket s is governed by the protocol Φ. In particular, if we have $s \Mapsto^{\mathsf{prot}} \Phi$ and $s \Mapsto^{\mathsf{prot}} \Psi$, we can conclude that Φ and Ψ are the same protocol.

We support two kinds of sockets: static sockets and dynamic sockets. This distinction is abstract, it is only at the level of the logic and not the distributed system itself. Static sockets are those which have primordial protocols agreed upon before starting the system. The static protocols are primarily for addresses pointing to servers. By having a primordial protocol, any node in the system (including the server itself) know and must respect this protocol.

To support node modular reasoning in general we distinguish static and dynamic addresses. To this end, we use the proposition $\overset{f}{\mapsto}(A)$ which means the set of addresses in A are static and should have a fixed interpretation. The proposition $\overset{f}{\mapsto}(A)$ expresses knowledge without asserting ownership of resources. In Iris terminology, this proposition is *persistent*: $\overset{f}{\mapsto}(A) \dashv\vdash \overset{f}{\mapsto}(A) * \overset{f}{\mapsto}(A)$.

Corresponding to the two kinds of addresses, we have the two rules Bind-stat-rule and Bind-dyn-rule shown below for binding addresses (starting the communication over) to a socket. Notice that in the case of Bind-dyn-rule one can choose the protocol while in the case of Bind-stat-rule the protocol is existentially quantified, *i.e.*, it is the primordial protocol associated to (ip, p).

```
rec lockserver ip p :=
  let lock := ref NONE in
  let skt := socket() in
  Socketbind skt (makeaddress ip p);
  listen skt (rec h msg from :=
    if msg = "LOCK"
    then match !lock with
         NONE => lock ← SOME ();
                 sendto skt "YES" from
       | SOME _ => sendto skt "NO" from
         end
    else lock ← NONE;
         sendto skt "RELEASED" from);
    listen skt h)
```

```
rec listen skt handler :=
  match receivefrom skt with
    SOME m => handler (fst m)
                      (snd m) in
  | NONE => listen skt handler
  end
```

Figure 51: A lock server in AnerisLang. Function binders are strings in Coq but are shown as regular binders for the sake of clarity.

BIND-STAT-RULE

$$\left\{\overset{f}{\mapsto}(A) * (ip,p) \in A * z \overset{s}{\mapsto}^{[n]} \mathsf{None} * \mathit{freePorts}(ip,\{p\})\right\}$$
$$\langle n; \mathsf{socketbind}\, z\, (ip,p)\rangle$$
$$\left\{x.\ x = 0 * \exists \Phi.\ z_n \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip,p) * (ip,p) \Mapsto^{\mathsf{prot}} \Phi\right\}$$

BIND-DYN-RULE

$$\left\{\overset{f}{\mapsto}(A) * (ip,p) \notin A * z_n \overset{s}{\mapsto}^{[n]} \mathsf{None} * \mathit{freePorts}(ip,\{p\})\right\}$$
$$\langle n; \mathsf{socketbind}\, z\, (ip,p)\rangle$$
$$\left\{x.\ x = 0 * z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip,p) * (ip,p) \Mapsto^{\mathsf{prot}} \Phi\right\}$$

Here, $z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip,p)$ is a socket assertion used to keep track of resources associated with the socket that belongs to node n. The resource is there to ensure that each socket is bound only once.

A Lock Server

Mutual exclusion in distributed systems is often a necessity and there are many different approaches for providing it. The simplest solution, presented in this section, is a centralized algorithm with a single node acting as a coordinator. The code is shown in Figure 51. We later show a more involved example of log-replication using two-phase-commit (5.7).

The lock server declares a node-local variable `lock` to keep track of the lock. It then binds a new socket `skt` on the given address `ip:p` and continuously listens for incoming messages on the socket. When a "LOCK" message arrives

and the lock is available, the lock is taken and the server responds "YES". If the lock was already taken the server responds with "NO". Finally, if the request is not "LOCK", the lock is released and the server responds with "RELEASED".

Notice that the lock server program looks as if it was written in a decent functional language with sockets. Messages sent and received are strings to make programming with sockets easier (similar to send_substring in the Unix module in OCaml). This is a direct result of the ambition to make AnerisLang useful for verifying existing code but also writing new programs in AnerisLang.

A node-local specification for the lock server, with a universally quantified lock source R and a protocol governing the socket endpoint $(ip, p) \Mapsto^{\mathsf{prot}} \phi$, is as follows:

$$\begin{array}{l}\left\{R * (ip,p) \Mapsto^{\mathsf{prot}} \phi * \overset{f}{\mapsto} (\{(ip,p)\} \cup A) * \mathit{freePorts}(ip,\{p\})\right\} \\ \quad \langle n; \mathsf{lockserver}\,() \rangle \\ \{\mathsf{True}\}\end{array}$$

There are several interesting observations one can make on the lock server example:

- The lock server can allocate, read and write node-local references but these are hidden in the specification.
- Sockets can be created and bound to specified endpoints. In this example we expect the lock server to be primordial, *i.e.*, the system should agree on a protocol $(ip, p) \Mapsto^{\mathsf{prot}} \phi$. Notice as well that there are no channel descriptors or assertions on the socket in the code.
- Without a proper protocol, the lock server fails to provide mutual exclusion since everyone can release the lock.

With the necessity of protocols on sockets established, we explain Aneris' interpretation of sockets and protocols in more detail.

Aneris Sockets and Protocols

Conceptually, a socket is an abstract representation of a handle for a local endpoint of some channel. In Aneris we further restrict channels to use the User Datagram Protocol (UDP), which is *asynchronous*, *connectionless* and *stateless*. In accordance with UDP, Aneris provides no guarantee of delivery or ordering, although we assume duplicate protection, since spatial resources could otherwise potentially be duplicated. One can therefore think of Aneris sockets as open-ended multi-party communication channels without synchronization.

It is noteworthy that inter-process communication can happen in multiple ways in Aneris. Thread-concurrent programs can communicate through the

store but they can also communicate by sending messages through sockets. For memory separated programs, there is no shared state and all communication is done with message-passing through sockets.

As mentioned in the Introduction, Iris has an un-opinionated approach to protocol design, therefore one can use a state-transition system if desired. However, modeling the execution of concurrent code with asynchronous message passing as state transition systems on thread and node levels can be quite cumbersome as opposed to a more descriptive approach in the form of resource reasoning.

A simple, novel approach to protocol design is to just give an Aneris predicate over messages to be received on the sockets. This can be regarded as a socket-local approach by requiring each socket to be guarded by an Aneris predicate. One can think of this as rely-reasoning, restricting the distributed environment's interference with a node on a particular socket. Ultimately, this is what allows node-local verification of programs.

Concretely, the socket protocol for a lock server can be specified as follows:

$$\begin{aligned} lock(m,\phi) &\triangleq body(m) = \text{"LOCK"} * \\ &\quad ((\forall m'.\, body(m') = \text{"NO"} \vee body(m') = \text{"YES"} * R) \mathrel{-\!\!*} \phi(m')) \\ rel(m,\phi) &\triangleq body(m) = \text{"RELEASE"} * R * \\ &\quad (\forall m'.\, body(m') = \text{"RELEASED"} \mathrel{-\!\!*} \phi(m)) \\ lock\_si &\triangleq \lambda m. \exists \phi.\, from(m) \Mapsto^{\mathsf{prot}} \phi * (lock(m,\phi) \vee rel(m,\phi)) \end{aligned}$$

The resources describing the lock, R, are transferred to the client if the server responds "YES" and the same resources must be returned when calling "RELEASE". The protocol is sufficient to prove that the clients of distributed system will use the lock server correctly.

Additionally, the lock protocol also illustrates how primordial servers respond to dynamic bound sockets. As mentioned earlier, the lock server socket should be primordial, however, the lock does not need to know about its clients as long as the clients follow the socket protocol defined by the lock server. As a consequence, a client has to prove that she can receive a reply from the server. This is specifically done by proving the resource-aware implication $\mathrel{-\!\!*}$ known as magic wand (expanded upon in 5.4).

The lock server protocol is very similar to a non-blocking specification of a locking module in concurrent separation logic, where the implication in the socket protocol can be seen as the post-condition.

5.3 Operational Semantics of AnerisLang

In this section, we present AnerisLang, a feature-rich concurrent programming language with network primitives. Usually, in a concurrent programming language, the operational semantics of the program are defined as a reduction

$$
\begin{aligned}
v &::= () \mid \mathsf{true} \mid \mathsf{false} \mid i \mid s \mid \ell \mid z \mid \mathsf{rec} f(x) = e \mid (v, v) \mid \mathsf{inj}_i\, v \mid \mathsf{address}\ s\ p \\
e &::= v \mid \boxdot e \mid e \circledcirc e \mid e\ e \mid \mathsf{find}\ e\ e\ e \mid \mathsf{substring}\ e\ e\ e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \\
&\quad \mid (e, e) \mid \pi_i\, e \mid \mathsf{inj}_i\, e \mid \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_i\, x \Rightarrow e_i\, \mathsf{end} \mid \mathsf{ref}(e) \mid\ !e \mid e \leftarrow e \\
&\quad \mid \mathsf{cas}(e, e, e) \mid \mathsf{fork}\ \{e\} \mid \mathsf{start}\ \{n; e; e\} \mid \mathsf{makeaddress}\, e\, e \mid \mathsf{socket} \\
&\quad \mid \mathsf{socketbind}\, e\, e \mid \mathsf{sendto}\, e\, e\, e \mid \mathsf{receivefrom}\, e
\end{aligned}
$$

Figure 52: Syntax of AnerisLang

relation over configurations consisting of pairs of a state, *i.e.*, heap, and a thread pool. AnerisLang is a programming language designed to model, reason about and implement distributed systems. These are systems comprised of a number of nodes, each of which concurrently runs a number of threads. Hence, configurations of AnerisLang are pairs consisting of state, *i.e.*, a heap for each node in the system and the state of the network, together with a collection of thread pools, one for each node.

In the sequel, we first present the syntax of AnerisLang. Subsequently, we show a *head-step* relation for programs. This head-step relation is then lifted to arbitrary AnerisLang configurations in the usual way in three steps: (1) a program takes a step if one of its nodes takes the corresponding step, (2) a node takes a step if one of its concurrently running threads does, and (3) a thread makes a step if the sub-expression of the program running on that thread in the evaluation position takes a head-step.

Syntax of AnerisLang

AnerisLang is a call-by-value, higher-order, concurrent imperative programming language with higher-order mutable references, fine-grained concurrency and network sockets. The syntax for values and expressions is shown in Figure 52.

In this figure v ranges over values and e ranges over expressions. In addition to the standard literal values we write i for integers and s for strings. The value $\mathsf{address}\ s\ p$ is a network address where s is the ip-address and p is the port number. Booleans, integers and strings can be manipulated by unary operations $\boxdot$ (negation, unary minus, string-length and conversions between strings and integers) and binary operations $\circledcirc$ (arithmetic operations, comparisons and test for equality). The string operations find and $\mathsf{substring}$ find the index of a particular substring and split a string producing a substring respectively.

We use ℓ for memory locations. These can be allocated by ref, read by $!l$, and updated by $l \leftarrow v$. The expression $\mathsf{fork}\ \{e\}$ forks off a new thread on the node it is running on. The atomic *compare-and-set* operation, $\mathsf{cas}(l, v_1, v_2)$, is used to achieve synchronization between threads on a specific memory

$$
\begin{aligned}
n \in Node, Ip &\triangleq String \\
a \in Address &\triangleq Ip \times Port \\
MessageState &\triangleq \{\textsc{Sent}, \textsc{Received}\} \\
Message &\triangleq Address \times Address \times String \times MessageState \\
MessageStable &\triangleq Address \times Address \times String \\
h \in Heap &\triangleq Loc \xrightarrow{\text{fin}} Val \\
S \in Sockets &\triangleq Handle \xrightarrow{\text{fin}} Address^? \\
Lookup &\triangleq Address \xrightarrow{\text{fin}} Node \\
Ports &\triangleq Ip \xrightarrow{\text{fin}} \wp^{\text{fin}}(Port) \\
MessageSoup &\triangleq MessageId \xrightarrow{\text{fin}} Message \\
\sigma \in NetworkState &\triangleq Node \xrightarrow{\text{fin}} Heap \times Node \xrightarrow{\text{fin}} Sockets \times Lookup \times Ports \times \\
&\qquad MessageSoup
\end{aligned}
$$

Figure 53: The physical state interpretation of Aneris

location l. Operationally, it does the following *atomically*: it tests whether ℓ has value v_1 and if so, updates the location to v_2. It returns true if successful and false otherwise.

The expression start $\{n; ip; e\}$ starts a new node n running program e assigned the ip-address ip. The only stipulation here is that nodes can only be started at the bootstrapping phase of the distributed system by a special system-node $\mathcal{S}$.

We use z to range over socket handles. The network primitives socket, socketbind, sendto and receivefrom correspond to the BSD-sockets API methods socket, to create sockets, bind, to bind to a socket, sendto, to send over the network on a socket, and, recvfrom, to receive over the network on socket, respectively. The expression makeaddress is used to compute addresses. The expression makeaddress $e\, e'$ evaluates to a network address if e evaluates to a string for the ip address and e' evaluates to a number for the port number [2].

Semantics of AnerisLang

We define the operational semantics of AnerisLang in three stages. First we define a node-local thread-local, head-step relation between configurations $(e, h, S) \rightarrow_{n,h} (e', h', S')$ for a heap h, allocated socket-endpoints S and expressions e, a node n and evaluation context $K \in Ectx$. Heaps are finite maps

[2] In the setting of OCaml, makeaddress would internally call `inet_addr_of_string`

from locations to values and sockets are finite maps from socket handles to option socket address. We write $K[e]$ to denote the expression of plugging e into the K and $e[v/x]$ to denote capture-avoiding substitution of v for x in e. Evaluation contexts are listed in Figure 54 and an excerpt of the node-local rules is shown in Figure 55.

$$
\begin{aligned}
K ::= {} & [\,] \mid \boxdot\, K \mid K \circledcirc e \mid v \circledcirc K \mid K\ e \mid v\ K \mid \mathsf{find}\ K\ e\ e \mid \mathsf{find}\ v\ K\ e \\
& \mid \mathsf{find}\ v\ v\ K \mid \mathsf{substring}\ K\ e\ e \mid \mathsf{substring}\ v\ K\ e \mid \mathsf{substring}\ v\ v\ K \\
& \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (K, e) \mid (v, K) \mid \pi_i\ K \mid \mathsf{inj}_i\ K \\
& \mid \mathsf{match}\ K\ \mathsf{with}\ \mathsf{inj}_i\ x \Rightarrow e_i\ \mathsf{end} \mid \mathsf{ref}(K) \mid\ !K \mid K \leftarrow e \mid v \leftarrow K \\
& \mid \mathsf{cas}(K, e, e) \mid \mathsf{cas}(v, K, e) \mid \mathsf{cas}(v, v, K) \mid \mathsf{makeaddress}\ K\ e \\
& \mid \mathsf{makeaddress}\ v\ K \mid \mathsf{socketbind}\ K\ e \mid \mathsf{socketbind}\ v\ K \mid \mathsf{sendto}\ K\ e\ e \\
& \mid \mathsf{sendto}\ v\ K\ e \mid \mathsf{sendto}\ v\ v\ K \mid \mathsf{receivefrom}\ K
\end{aligned}
$$

Figure 54: Evaluation contexts of AnerisLang

Node-local steps are lifted to a network-aware stepping relation $\rightarrow_h$ by lifting expressions to node-expressions $\epsilon = \langle n; e \rangle$ and lifting the local state $\mathit{Heap} \times \mathit{Sockets}$ to $\mathit{NetworkState}$, tracking heaps H and sockets Z for all nodes, all bound addresses in the system L, ports in use P and messages sent M. The network-aware stepping relation is shown in Figure 56.

$$
((\mathsf{rec} f(x) = e)\ v, h, S) \rightarrow_{n,h} (e[v, (\mathsf{rec} f(x) = e)/x, f], h, S)
$$

$$
(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, h, S) \rightarrow_{n,h} (e_2, h, S) \qquad (\pi_1\ (v_1, v_2), h, S) \rightarrow_{n,h} (v_1, h, S)
$$

$$
\frac{\ell \notin \mathrm{dom}(h)}{(\mathsf{ref}(v), h, S) \rightarrow_{n,h} (\ell, h \uplus \{\ell \mapsto v\}, S)} \qquad \frac{h = h' \uplus \{\ell \mapsto v'\}}{(\ell \leftarrow v, h, S) \rightarrow_{n,h} ((), h' \uplus \{\ell \mapsto v\}, S)}
$$

$$
\frac{v = h(\ell)}{(!\,\ell, h, S) \rightarrow_{n,h} (v, h, S)} \qquad \frac{h = h' \uplus \{\ell \mapsto v\}}{(\mathsf{cas}(\ell, v, v'), h, S) \rightarrow_{n,h} (\mathsf{true}, h' \uplus \{\ell \mapsto v'\})}
$$

$$
\frac{h = h' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\mathsf{cas}(\ell, v, v'), h, S) \rightarrow_{n,h} (\mathsf{false}, h, S)} \qquad \frac{h = h' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\mathsf{cas}(\ell, v, v'), h, S) \rightarrow_{n,h} (\mathsf{false}, h, S)}
$$

$$
\frac{z \notin \mathrm{dom}(S)}{(\mathsf{socket}, h, S) \rightarrow_{n,h} (z, h, S \uplus \{z \mapsto \mathsf{None}\})}
$$

Figure 55: An excerpt of the node-local head-reduction rules

$$\frac{(e,h,S) \to_{n,h} (e',h',S') \qquad Z' = Z[n \mapsto S'] \qquad H' = H[n \mapsto h']}{\langle n;e\rangle,(H[n \mapsto h],Z[n \mapsto S],L,P,M) \to_h \langle n;e'\rangle,(H',Z',L,P,M)}$$

$$\frac{\begin{array}{c}Z(n) = S \qquad S(z) = \mathsf{None} \qquad (ip,p) = a \qquad a \notin \mathrm{dom}(L) \qquad p \notin P(ip) \\ Z' = Z[n \mapsto S[z \mapsto \mathsf{Some}\, a]] \qquad L' = L[a \mapsto n] \qquad P' = P[a \mapsto P(a) \cup \{p\}]\end{array}}{\langle n;\mathsf{socketbind}\, z\, a\rangle,(H,Z,L,P,M) \to_h \langle n;0\rangle,(H,Z',L',P',M)}$$

$$\frac{\begin{array}{c}Z(n)(z) = \mathsf{Some} from \qquad m = (from,to,msg,\textsc{Sent}) \qquad m_{id} \notin dom(M) \\ M' = M[m_{id} \mapsto m]\end{array}}{\langle n;\mathsf{sendto}\, z\, msg\, to\rangle,(H,Z,L,P,M) \to_h \langle n;\mathsf{length}\ msg\rangle,(H,Z,L,P,M')}$$

$$\frac{\begin{array}{c}Z(n)(z) = \mathsf{None} \qquad (ip,p) \notin dom(L) \qquad p \notin P(ip) \qquad m_{id} \notin dom(M) \\ m = ((ip,p),to,msg,\textsc{Sent}) \qquad M' = M[m_{id} \mapsto m]\end{array}}{\langle n;\mathsf{sendto}\, z\, msg\, to\rangle,(H,Z,L,P,M) \to_h \langle n;\mathsf{length}\ msg\rangle,(H,Z,L,P,M')}$$

$$\frac{\begin{array}{c}Z(n)(z) = \mathsf{Some}\, a \qquad m_{id} \mapsto m \in M \qquad from(m) = f \qquad to(m) = a \\ msg(m) = b \qquad state(m) = \textsc{Sent} \\ m' = (f,a,b,\textsc{Received}) \qquad M' = M[m_{id} \mapsto m']\end{array}}{\langle n;\mathsf{receivefrom}\, z\rangle,(H,Z,L,P,M) \to_h \langle n;\mathsf{Some}\,(b,f)\rangle,(H,Z,L,P,M')}$$

$$\frac{Z(n)(z) = \mathsf{Some}\, a \qquad \emptyset = \{m_{id} \,|\, m_{id} \mapsto (-,a,-,\textsc{Sent}) \in M\}}{\langle n;\mathsf{receivefrom}\, z\rangle,(H,Z,L,P,M) \to_h \langle n;\mathsf{None}\rangle,(H,Z,L,P,M)}$$

Figure 56: Network-aware head-reduction rules for sockets.

The rule for socketbind expresses that a socket can be bound if it is not already bound, the address is not already used in the network and the port is not in use at the address. Hereafter, the address is stored in the lookup table L and the port is no longer available in P. One can think of L as a global name-server making sure address-conflicts do not occur.

There are two rules for sending messages through a sockets by sendto – one for bound sockets, $S(s) = \mathsf{Some}\, a$, and one for unbound sockets, $S(s) = \mathsf{None}$. For bound sockets, the message is populated with the sender's address (*from*), a destination address (*to*), the message itself and a status flag indicating sent (and not received). The operation returns the number of characters sent. For the unbound rule one has to show a free address exists. Notice that we do not care if the destination exists or not.

Finally, messages can be received by the receivefrom rule. A socket can receive messages if it has a registered address and awaiting messages. Upon receiving the message, the message and sender are returned and the status

flag of the message is updated to Received. If no messages are waiting, None is returned.

The final head-step relation is a *distributed systems* relation $\rightarrow$ shown below. The distributed systems relation reduces by picking a thread on any node or forking off a new thread on a node.

$$\frac{(\langle n; e\rangle, \sigma) \rightarrow_h (\langle n; e'\rangle, \sigma')}{(\vec{\epsilon}_1, \langle n; K[e]\rangle, \vec{\epsilon}_2; \sigma) \rightarrow (\vec{\epsilon}_1, \langle n; K[e']\rangle, \vec{\epsilon}_2; \sigma')}$$

$$(\vec{\epsilon}_1, \langle n; K[\mathsf{fork}\ \{e\}]\rangle, \vec{\epsilon}_2; \sigma) \rightarrow (\vec{\epsilon}_1, \langle n; K[()]\rangle, \vec{\epsilon}_2, \langle n; e\rangle; \sigma)$$

5.4 Semantics of Aneris

With the Semantics of AnerisLang defined we focus on building the logic for Aneris in this section. We start by introducing the Iris logic before presenting the Aneris logic and stating adequacy.

A Primer on Iris

Iris was originally presented as a framework for higher-order (concurrent) separation logic inside the proof assistant Coq. It has a built-in notion of physical state, ghost-state (monoids) and invariants. Moreover, it includes weakest preconditions which are useful for Hoare-style reasoning of concurrent imperative programs [33]. Recently, a simpler Iris *base logic* was defined. This base logic suffices for defining all earlier built-in concepts, such as invariants, higher-order ghost state and weakest preconditions. [38]. It is this latter version of Iris we depend on for defining Aneris.

The quantifiable types of Iris, denoted by κ, are shown below:

$$\begin{aligned}\kappa ::= {} & 1 \mid \kappa \times \kappa \mid \kappa \rightarrow \kappa \mid \mathbb{N} \mid \mathbb{B} \mid \mathit{Ectx} \mid \mathit{Var} \mid \mathit{Expr} \mid \mathit{Val} \mid \kappa \xrightarrow{\text{fin}} \kappa \mid \mathsf{finset}(\kappa) \\ & \mid \mathit{Monoid} \mid \mathit{Names} \mid \mathit{iProp} \mid \ldots\end{aligned}$$

Iris includes basic types such as the unit, 1, pairs and ordinary functions. $\mathbb{N}$ is the type of natural numbers, $\mathbb{B}$ is the type of booleans and *Expr* and *Val* are the types of AnerisLang syntactic expressions and values. The type $\kappa \rightharpoonup^{\text{fin}} \kappa$ is that of partial functions with finite support and $\mathsf{finset}(\kappa)$ is the type of finite sets. *Monoid* is the type of monoids, *Names* is the type of ghost names, and *iProp* is the type of Iris propositions. An excerpt of the grammar for Iris propositions P is:

$$\begin{aligned}P ::= {} & \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \\ & \mid \Box P \mid \mathsf{wp}\, \langle n; e\rangle \left\{x.\ P\right\} \mid \{P\}\, \langle n; e\rangle\, \{x.\ Q\} \mid \Rrightarrow P \mid \boxed{P}^{\mathcal{N}} \mid \ldots\end{aligned}$$

The grammar includes the usual connectives of higher-order separation logic ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $*$, $-\!\!*$, $\forall$ and $\exists$). In this grammar Φ is an Iris predicate, *i.e.*, a term of type $\kappa \to \mathit{iProp}$ (for an appropriate κ). Intuitively, as in any separation logic, propositions P denote sets of resources satisfied by P. Propositions joined by separating conjunction, $P * P'$, express that resources can be split into disjoint parts; one satisfying P and the other satisfying P'. Propositions $P -\!\!* P'$ describe those resources that, combined with disjoint resources satisfied by P, can result in resources satisfied by P'. In addition to these standard separation logic connectives, Iris includes other useful connectives of which the most frequently used ones will be described below.

Arguably, the most important feature of any separation logic is the ability to update the parts you own while leaving all other resources intact. In Iris this is accomplished by the "update" modality[3], $\Rrightarrow$. Intuitively, $\Rrightarrow P$ holds for those resources that can be updated to resources that satisfy P, without violating the environment's knowledge or ownership of resources. The update modality is idempotent, $\Rrightarrow(\Rrightarrow P) \dashv\vdash \Rrightarrow P$. We write $P \Rrightarrow Q$ as a shorthand for $P -\!\!* \Rrightarrow Q$.

Possibly, the second most important feature of any separation logic is the concept of invariants. Conceptually, invariants capture what is known, as opposed to what is owned. In Iris, the "persistence" modality ($\Box$) is used to capture a sub logic of knowledge that obeys standard rules for intuitionistic, higher-order logic. We say that a proposition is *persistent* if $P \vdash \Box P$. Intuitively, $\Box P$ are those propositions that are satisfied without asserting any exclusive ownership. Consequently, because $\Box P$ asserts no exclusive ownership, it is a duplicable assertion $(\Box P) * (\Box P) \dashv\vdash \Box P$. The persistence modality is idempotent, that is, $\Box P \vdash \Box\Box P$ for any P. Furthermore, $\Box P \vdash P$. The persistence modality also commutes with all of the connectives of higher-order separation logic.

The $\triangleright$ modality, pronounced "later", is an abstraction of step-indexing [5, 6, 21]. For any proposition P, we have that $P \vdash \triangleright P$, which in terms of step-indexing means that if P holds now then it also does so a step later. In Iris, the $\triangleright$ modality is used in the definition of weakest preconditions and to guard impredicative invariants to avoid self-referential paradoxes [38]. The later modality commutes with all of the connectives of higher-order separation logic.

The propositions $\mathsf{wp}\langle n; e\rangle\{x.\ P\}$ and $\{P\}\langle n; e\rangle\{x.\ Q\}$ are Iris's propositions for reasoning about programs, where Iris is instantiated with AnerisLang. Intuitively, $\mathsf{wp}\langle n; e\rangle\{x.\ P\}$ holds when the expression e is *safe* to execute, *i.e.*, it does not get stuck, and that whenever it reduces to a value v, then $P[v/x]$ holds. In Iris, Hoare triples are defined based on weakest preconditions in the

[3] In [38] this modality is called the *fancy* update modality. Technically, this modality comes equipped with certain "masks" but we do not discuss those here.

usual manner:

$$\{P\}\langle n; e\rangle\{x.\ Q\} \triangleq \Box\Big(P \mathrel{-\!\!*} \mathsf{wp}\,\langle n; e\rangle\Big\{x.\ Q\Big\}\Big)$$

Notice that we require Hoare triples to be persistent, *i.e.*, Hoare triples assert only knowledge and no ownership of resources. In other words, the Hoare triple $\{P\}\langle n; e\rangle\{x.\ Q\}$ states that all the (non-persistent) resources that are used by e are contained in P. At any point in the course of proving correctness of a program we can update resources. This is captured in the following property of the weakest preconditions:

$$\Rightarrow\!\!\Vdash \mathsf{wp}\,\langle n; e\rangle\Big\{x.\ P\Big\} \dashv\vdash \mathsf{wp}\,\langle n; e\rangle\Big\{x.\ P\Big\} \dashv\vdash \mathsf{wp}\,\langle n; e\rangle\Big\{x.\ \Rightarrow\!\!\Vdash P\Big\}$$

The weakest precondition for values is equivalent to the postcondition holding for that value, modulo updating resources:

$$\textsc{wp-value}\quad \mathsf{wp}\,\langle n; v\rangle\Big\{x.\ P\Big\} \dashv\vdash \Rightarrow\!\!\Vdash P[v/x]$$

Proving safety of a program under an evaluation context, $K[e]$, can be done in two separate steps: (1) we prove that e is safe, and, (2) for a value w that we get out of the execution of e, $K[w]$ is safe to execute. This fact is embodied in Iris as the 5.4 rule below:

$$\textsc{wp-bind}\quad \frac{\mathsf{wp}\,\langle n; e\rangle\Big\{y.\ \mathsf{wp}\,\langle n; K[y]\rangle\Big\{x.\ P\Big\}\Big\}}{\mathsf{wp}\,\langle n; K[e]\rangle\Big\{x.\ P\Big\}}$$

Iris features invariants $\boxed{P}^{\mathcal{N}}$, pronounced invariantly P. The name $\mathcal{N}$ is the name associated with the invariant and it is used to keep track of which invariants are opened as opening invariants multiple times in a nested fashion is in general unsound.[4] Invariants are Iris's way of encoding protocols for shared resources. Invariants, once established, can only be violated during the execution of an atomic program step, *i.e.*, the time when it cannot be noticed by other threads. The following two rules allow us to establish and open invariants:

$$\textsc{inv-alloc}\quad \frac{\triangleright P}{\Rightarrow\!\!\Vdash \boxed{P}^{\mathcal{N}}} \qquad \textsc{inv-open}\quad \frac{\boxed{P}^{\mathcal{N}} \qquad \triangleright P \mathrel{-\!\!*} \mathsf{wp}\,\langle n; e\rangle\Big\{x.\ \triangleright P * Q\Big\} \qquad e \text{ is phys. atomic}}{\mathsf{wp}\,\langle n; e\rangle\Big\{x.\ Q\Big\}}$$

Note the use of the later modality with invariants. This is necessary as invariants are impredicative, *i.e.*, $\boxed{P}^{\mathcal{N}}$ is an *iProp* and so is P. The later modality

[4] Indeed the update modality is annotated with a mask consisting of the set of names of invariants that are opened before and after the update to prevent such unsound opening of invariants. We omit masks of the update modality in this paper for the sake of brevity and simplicity.

is necessary [38] to ensure that invariants do not allow encoding of self-referential paradoxes. Invariants are persistent as they simply assert the knowledge that some proposition holds invariantly.

Aneris: the program logic

Iris as a (program) logic consists of two layers: the Iris base logic, explained above in Subsection 5.4 except for weakest preconditions and Hoare triples, and a program logic defined on top of the base logic. The program logic layer of Iris is language agnostic. That is, the user of Iris can specify the syntax and operational semantics of their programming language together with the resources one needs to keep track of the state of programs (the heaps of individual nodes and the state of the network in our case) and as a result get a program logic, *i.e.*, a basic notion of weakest preconditions for that language. This is what we have done for Aneris: we have instantiated Iris's program logic layer with the syntax and operational semantics of Aneris given in Section 5.3.

The basic notion of weakest preconditions provided by the program logic layer of Iris is defined using the Iris base logic, based on the given operational semantics. In order to get a full-blown program logic one needs to derive the intended rules of the logic based on the definition of weakest preconditions provided by Iris.[5] In the sequel we present the weakest precondition rules that we derive for Aneris. We use these rules for proving correctness of the programs that we discuss in this paper.

Aneris program logic rules can be divided into three classes: those pertaining to pure computations, those for manipulating the heap (of a node) and those for network-communications. We discuss them in that order.

The weakest precondition rules for pure computations are exactly as one would expect. An instructive excerpt is given below:

$$\textsc{fst-wp}\ \frac{\triangleright\, \mathsf{wp}\,\langle n; v\rangle\{\Phi\}}{\mathsf{wp}\,\langle n; \pi_1\,(v,w)\rangle\{\Phi\}} \qquad \textsc{if-true-wp}\ \frac{\triangleright\, \mathsf{wp}\,\langle n; e\rangle\{\Phi\}}{\mathsf{wp}\,\langle n; \mathsf{if\,true\,then}\,e\,\mathsf{else}\,e'\rangle\{\Phi\}}$$

$$\textsc{rec-wp}\ \frac{\triangleright\, \mathsf{wp}\,\langle n; e[\mathsf{rec}\,f(x) = e, v/f, x]\rangle\{\Phi\}}{\mathsf{wp}\,\langle n; (\mathsf{rec}\,f(x) = e)\ v\rangle\{\Phi\}}$$

Note that all these programs take a step of computation to execute and hence their antecedent is only required to hold at a later step.

[5] Iris as a program logic framework comes with an internal programming language. For this programming language, the Iris program logic is instantiated and all the relevant program logic rules are derived. Here, we put this internal language aside and start from scratch: we define syntax and semantics of Aneris and derive all the program logic proof rules presented in this section.

The weakest preconditions for heap-manipulating programs are similar to their analogues in standard separation logic for ML-like programming languages:

$$\textsc{wp-alloc}\quad \frac{\forall \ell.\ \ell \mapsto^{[n]} v \mathrel{-\!\!*} \mathsf{wp}\,\langle n; \ell\rangle\{\Phi\} \qquad \triangleright \mathit{IsNode}(n)}{\mathsf{wp}\,\langle n; \mathsf{ref}(v)\rangle\{\Phi\}}$$

$$\textsc{wp-load}\quad \frac{\ell \mapsto^{[n]}_q v \mathrel{-\!\!*} \mathsf{wp}\,\langle n; v\rangle\{\Phi\} \qquad \triangleright \ell \mapsto^{[n]}_q v}{\mathsf{wp}\,\langle n; !\ell\rangle\{\Phi\}}$$

$$\textsc{wp-store}\quad \frac{\ell \mapsto^{[n]} w \mathrel{-\!\!*} \mathsf{wp}\,\langle n; ()\rangle\{\Phi\} \qquad \triangleright \ell \mapsto^{[n]} v}{\mathsf{wp}\,\langle n; \ell \leftarrow w\rangle\{\Phi\}}$$

$$\textsc{wp-cas-suc}\quad \frac{\ell \mapsto^{[n]} w \mathrel{-\!\!*} \mathsf{wp}\,\langle n; \mathsf{true}\rangle\{\Phi\} \qquad \triangleright \ell \mapsto^{[n]} v}{\mathsf{wp}\,\langle n; \mathsf{cas}(\ell, v, w)\rangle\{\Phi\}}$$

$$\textsc{wp-cas-fail}\quad \frac{\ell \mapsto^{[n]} v' \mathrel{-\!\!*} \mathsf{wp}\,\langle n; \mathsf{false}\rangle\{\Phi\} \qquad \triangleright \ell \mapsto^{[n]} v' \qquad v \neq v'}{\mathsf{wp}\,\langle n; \mathsf{cas}(\ell, v, w)\rangle\{\Phi\}}$$

The propositions $\ell \mapsto^{[n]} v$ and $\ell \mapsto^{[n]}_q v$ are called points-to and fractional points-to propositions respectively. These are similar to their counterparts in other (concurrent) separation logics; they are simply annotated with the node they belong to. They both assert that a memory location ℓ in the heap of node n has value v. The former asserts full ownership of this location while the latter asserts only the ownership of a fraction $0 < q \leq 1$ of this memory location. In particular, we have $\ell \mapsto^{[n]}_1 v \dashv\vdash \ell \mapsto^{[n]} v$. The proposition $\mathit{IsNode}(n)$ indicates that the node n is a valid node in the system. It is required for allocation of new memory locations but not for reading and updating them. This is because having a (fractional) points-to proposition for a location on a node in and of itself is indicative of the fact that that node is a valid node.

$$\textsc{wp-socket}\quad \frac{\forall z.\ z \overset{s}{\mapsto}^{[n]} \mathsf{None} \mathrel{-\!\!*} \mathsf{wp}\,\langle n; z\rangle\{\Phi\} \qquad \triangleright \mathit{IsNode}(n)}{\mathsf{wp}\,\langle n; \mathsf{socket}\rangle\{\Phi\}}$$

WP-SOCKET-BIND-DYN

$$\frac{\forall g.\ z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip,p) * (ip,p) \overset{r}{\mapsto} g * (ip,p) \Rightarrow^{\mathsf{prot}} \phi \mathrel{-\!\!*} \mathsf{wp}\,\langle n;0\rangle\{\Phi\} \qquad \mathit{freePorts}(ip,\{p\}) \qquad \overset{f}{\mapsto} A \qquad (ip,a) \notin A \qquad \phi \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{None}}{\mathsf{wp}\,\langle n;\mathsf{socketbind}\,z\,(ip,a)\rangle\{\Phi\}}$$

WP-SOCKET-BIND-STAT

$$\frac{\forall g.\ z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip,p) * (ip,p) \overset{r}{\mapsto} g * \mathrel{-\!\!*} \mathsf{wp}\,\langle n;0\rangle\{\Phi\} \qquad \overset{f}{\mapsto} A \qquad \mathit{freePorts}(ip,\{p\}) \qquad (ip,p) \in A \qquad (ip,p) \Rightarrow^{\mathsf{prot}} \phi \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{None}}{\mathsf{wp}\,\langle n;\mathsf{socketbind}\,z\,(ip,a)\rangle\{\Phi\}}$$

WP-SEND-TO-BOUND

$$\frac{P \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,a \qquad d \Rightarrow^{\mathsf{prot}} \phi \qquad \forall m_{id}, M.\ \mathit{mSoup}(M) * m_{id} \overset{st}{\mapsto} (a,d,s) * P \Rrightarrow \mathit{mSoup}(M) * \triangleright \phi(a,d,s) * Q \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,a * Q \mathrel{-\!\!*} \mathsf{wp}\,\langle n;\mathit{length}(s)\rangle\{\Phi\}}{\mathsf{wp}\,\langle n;\mathsf{sendto}\,z\,s\,d\rangle\{\Phi\}}$$

WP-SEND-TO-UNBOUND

$$\frac{P \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{None} \qquad d \Rightarrow^{\mathsf{prot}} \phi \qquad a = (ip,p) \qquad \mathit{freePorts}(ip,\{p\}) \qquad \forall p, m_{id}, M.\ \mathit{mSoup}(M) * m_{id} \overset{st}{\mapsto} (a,d,s) * P \Rrightarrow \mathit{mSoup}(M) * \triangleright \phi(a,d,s) * Q \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{None} * Q \mathrel{-\!\!*} \mathsf{wp}\,\langle n;\mathit{length}(s)\rangle\{\Phi\}}{\mathsf{wp}\,\langle n;\mathsf{sendto}\,z\,s\,d\rangle\{\Phi\}}$$

WP-RECEIVE-FROM

$$\frac{z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,a \qquad a \overset{r}{\mapsto} g \qquad a \Rightarrow^{\mathsf{prot}} \phi \qquad N \triangleq z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,a * a \overset{r}{\mapsto} g \qquad r(m) \triangleq (\mathit{body}(m), \mathit{from}(m)) \qquad S(m_{id},m) \triangleq z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,a * a \overset{r}{\mapsto} \{m_{id} \mapsto m\} \cup g * m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m * \phi(\mathit{stable}(m)) \qquad N \mathrel{-\!\!*} \mathsf{wp}\,\langle n;\mathsf{None}\rangle\{\Phi\} \vee \exists m_{id}, m.\ S(m_{id},m) \mathrel{-\!\!*} \mathsf{wp}\,\langle n;\mathsf{Some}\,r(m)\rangle\{\Phi\}}{\mathsf{wp}\,\langle n;\mathsf{receivefrom}\,z\rangle\{\Phi\}}$$

Similar to allocation of locations on the heap, to allocate a socket by WP-SOCKET, one must provide $\mathit{IsNode}(n)$ to indicate that n is valid. A socket points to $z \overset{s}{\mapsto}^{[n]} \mathsf{None}$ is returned, that enjoy the same properties as those for heaps.

We already touched upon binding of sockets to both primordial and dynamic addresses in 5.2, but here we present the actual rule as defined in Aneris. What is new is the ghost-assertion $(ip,p) \overset{r}{\mapsto} g$ which is a necessary evil if one wants to build protocols on top of the UDP communication channel. If we only used the $z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip,p)$ we could not prove that no other messages

was received on the socket. Notice again, that the dynamic bind allows one to use a custom socket protocol ϕ.

When a socket has been bound to an address a, the duplicable assertion $a \Mapsto^{\mathsf{prot}} \phi$ is returned, stating, that all participants in the distributed system should obey the socket protocol.

Arguable, the most interesting part is the interaction with the socket protocols in sendto and receivefrom. For sending, one has to prove, via $\Rrightarrow\!\!*$, that the resources described by ϕ can be obtained from the resources one currently owns and the message stable resource $m_{id} \overset{st}{\mapsto} (a, d, s)$, conveying that a send operation took place. The P and Q are there to allow the client to prove custom protocols at the atomic place the send operation occurs. When complete, the resources guarding the socket protocol are transferred to the network while the messages are in transit.

UDP sockets allow for both sending messages through bound sockets and unbound sockets. For unbound sockets, an available socket will be picked during the atomic send and released immediately afterwards. To allow for picking an available port, the $\mathit{freePorts}(ip, \emptyset)$ has to be provided. Replies on unbound ports is technically possible, however, logically one would know the socket protocol for the sender address.

When calling receivefrom one of two things can happen; either you will receive a message or no messages are available on the socket. If a message can be received, the resources described by ϕ are now transferred from the network to the node owning the socket. Furthermore, a certificate that says the message has been received $m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m$ is also returned to the client. The fractional permission that the environment retain ensures that the client cannot change the state post-delivery.

It may not be completely obvious that one can encode protocols that require progress by these resources, however, consider the following socket protocol.

$$\begin{aligned}\phi(p)(m) \triangleq \exists a, m_{id}, n, \phi'.\, & m_{id} \overset{st}{\mapsto} (a, p, n) * a \Mapsto^{\mathsf{prot}} \phi' * \\ & (\forall m'_{id}.\, m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m * m'_{id} \overset{st}{\mapsto} (p, a, n+1) \mathrel{-\!\!*} \phi'(p, a, n+1))\end{aligned}$$

Here, a socket can require that when another participant q sends a node on a socket to p, q must be able to receive a message where the number has been incremented by 1. However, if the protocol did not assert $m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m$, p could cheat and potentially send a message ahead of time. With the resources in place, such cheating can be avoided.

Adequacy

A (program) logic is at most as good as its soundness/adequacy theorem. That is, one needs to answer the question: "what do we get when prove a theorem in the logic?". Concretely in our case, what, if anything, can we conclude from a proof of a weakest precondition proven in Aneris' logic? The short answer is *safety*. That is, when running a distributed system for which we have proven a weakest precondition, none of the nodes in the system crash. This property, *i.e.*, weakest preconditions implying safety, is stronger than meets the eye at first. The reason for this is, that it is possible, that violating socket protocols can cause a node to crash, *e.g.*, a client of the lock server in 5.2. However, proving weakest precondition proves safety, so it must imply that protocols are followed for all parties in the distributed system.

The adequacy theorem The statement of the adequacy theorem in our Coq development is the following:

```
Theorem adequacy `{distPreG Σ} (IPs : gset ip_address)
    (A : gset socket_address) e σ :
  (∀ `{distG Σ}, (|={⊤}=> ∃ (f : socket_address → socket_interp Σ),
            ↦^f A -★ ([★ set] a ∈ A, a ⤇^prot (f a)) -★
            ([★ set] ip ∈ IPs, FreeIP ip) -★ WP e {{v, True }})%I) →
  dom (gset ip_address) (state_ports_in_use σ) = IPs →
  (∀ i, i ∈ IPs → state_ports_in_use σ !! i = Some ∅) →
  state_heaps σ = ∅ → state_sockets σ = ∅ → state_lookup σ = ∅ →
  state_ms σ = ∅ → safe e σ.
```

Notice that the `%I` instructs Coq to parse logical connectives as Iris connectives instead of Coq connectives. The symbol `|={⊤}=>` is how we write the update modality in Coq where $\top$ is the mask of this modality for invariant names that we have omitted in this paper. We write `[★ set] a ∈ A` in Coq for the big separating conjunction connective.

This theorem is a bit long but its reading is straightforward: It proves that running Aneris program e is safe starting from state σ if the following conditions hold:

- That, under the assumption that resources are initialized, `distG Σ`, and given predefined socket protocols for all the primordial sockets, `A`, and having all necessary free ip-addresses, `IPs`, the weakest precondition for e holds in Aneris.
- In σ there are no heaps, no sockets, and the lookup table and messages are also empty
- The set of IP-address in σ should be exactly `IPs` and the set of ports used in each of the ip-addresses should be empty.

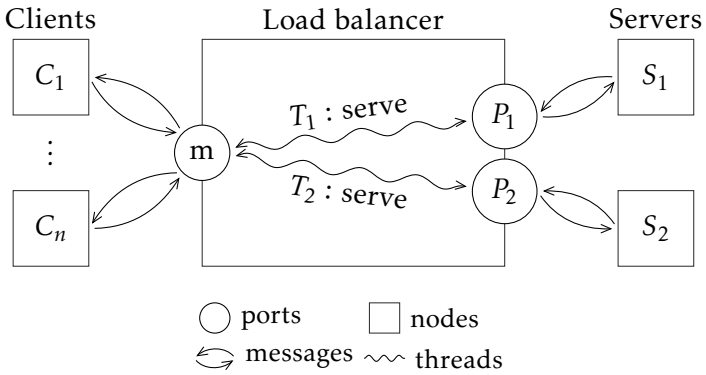


Figure 57: The architecture of a distributed system with a load balancer.

The `distPreG Σ` is the assumption stating that the set of resources that Aneris is parameterized with includes all necessary resources for initializing the distributed systems. Individual proofs of some distributed systems might require more specific resources than the basic resources provided. One has to show those resources can be allocated when proving adequacy of those particular systems.

The adequacy theorem above is a direct consequence of the adequacy theorem of Iris which says that closed (not relying on any particular resource) proofs of weakest preconditions imply safety. To prove the adequacy of Aneris we need to show that all the resources necessary for distributed systems can be initialized appropriately. In other words, we need to instantiate `distG Σ`.

Note that the final result of the adequacy theorem above is safety, a fact within Coq independent of Aneris and Iris. In other words, when we verify a program within Aneris, we get that the program is safe to execute independent of Aneris or Iris. So indeed one must only trust Coq as formal system and need not trust Iris or our program logic Aneris build on top of it.

5.5 Case Study 1: A Load Balancer

As mentioned earlier, AnerisLang supports concurrency through its fork $\{e\}$ primitive. One example for illustrating the benefit of this primitive is server-side load balancing. Load balancing is generally used in order to distribute workload in a distributed system and is commonly utilized with the goal of achieving horizontal scaling wrt. providing Internet services.

Implementation of a Load Balancing Protocol

In the case of server-side load balancing, the distribution of work is done by a program listening on a specific port that clients send their requests to. This

```
rec load_balancer ip port servers :=
 let main := socket() in
 socketbind skt (makeaddress ip port);
 list_fold (λ server acc :=
  fork (serve main ip acc server);
  acc + 1) 1100 servers

rec listen_wait skt :=
 match receivefrom skt with
   SOME m => m
 | NONE => listen_wait skt
 end
```

```
rec serve main ip port server :=
 let skt := socket() in
 socketbind skt (makeaddress ip port);
 (rec loop () :=
  match receivefrom main with
   SOME m =>
    let msg := Fst m in
    let sender := Snd m in
    sendto skt msg server in
    let res := Fst (listen_wait skt) in
    sendto main res sender;
    loop ()
 | NONE => loop ()
 end) ()
```

Figure 58: An implementation of a load balancer in AnerisLang.

program then sends the request to one of the available servers, waits for the answer from the server and sends this answer back to the client. In order to be able to handle requests from several clients simultaneously, the load balancer can employ concurrency by forking off a new thread for every available server in the system. Each of these threads will then listen for and handle requests. The architecture of such a system with two servers is illustrated in Figure 57.

As can be seen in Figure 58, the `load_balancer` module expects an ip address, a port and a list of servers. It then creates and binds to a socket with the given ip address and port. Finally, it folds over the list of servers, forking off a new thread for each server, running the `serve` module with the newly-created socket, the given ip address, a fresh port number and the current server as arguments.

The `serve` module expects a socket, an ip address, a port and a server address. It first creates and binds to a new socket with the given ip address and port number. After this, `serve` continuously tries to receive a message on the `main` socket. This message would be a request from a client. Once a request is received, it is passed on to the given server via the fresh socket and `serve` waits until it receives an answer from the server, which it finally passes on to the client via the `main` socket. This way the entire process is hidden from the client, whose view will be the same as if it was communicating with just a single server handling its request.

Specification and Protocols

In order to keep the specification of the load balancer as general as possible, we parameterize its socket protocol by the predicates $P_v : X \to iProp$, and P_{in} and P_{out} of type $MessageBody \to X \to iProp$ describing the client request and server response, respectively. All predicates is parameterized by the type X

which can be used for maintaining state between the request and the response. These predicates may not depend of the sender of messages since the sender changes when relaying the message forward from the load-balancer. The relay socket protocol is shown below:

$$\phi_{rel}(P_v, P_{in}, P_{out}) \triangleq \lambda m. \exists \varphi, (v : X). from(m) \Mapsto^{\mathsf{prot}} \varphi * P_{in}(body(m), v) * P_v(v) * (\forall m'. P_v(v) * P_{out}(body(m'), v) \mathrel{-\!\!*} \varphi(m'))$$

The load balancer and each of the servers are bound by the same socket protocol. Combined, this enables the `serve` module to relay requests from the `load_balancer` socket to the server and responses in the opposite direction without invalidating the socket protocol.

The socket protocol that clients communicate with will normally have $P_{\mathsf{True}} \triangleq \lambda v, \mathsf{True}$, thus the client only has to provide $P_{in}(s, v)$ when sending a message s to the service, via the load balancer. The request the `serve` module receives already fulfills $P_{in}(s, v)$, which is needed for passing the message along to the service, however, the load-balancer needs to make sure the server behaves faithfully when responding to requests. Otherwise, the serve module would not be able to fulfill the client's socket protocol. To this end, the server protocol must have the following P_v defined in its socket protocol, conceptually connecting the request to the server with the response from the server.

$$P_{v\_lb}(server) \triangleq \lambda(v : X), server \mapsto_{\frac{1}{2}} v$$

The socket protocol for the serve module can now be given as such:

$$\phi_{server}(server, P_{out}) \triangleq \lambda m. \exists v. a \mapsto_{\frac{1}{2}} v * P_{out}(body(m), v)$$

Since all instances of the `serve` module need to access the `main` socket in order to receive requests and send answers, we have to put the resources required for accessing a socket in an invariant which is shared by all threads.

$$lb\_inv(n, z, a) \triangleq \boxed{\exists g. z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\, a * a \overset{r}{\mapsto} g}^{\mathcal{N}}$$

The specification for the `serve` module is:

$$\left\{ \begin{array}{l} lb\_inv(n, main, ma) * \overset{f}{\mapsto} (A) * (ip, p) \notin A * freePorts(ip, \{p\}) * IsNode(n) * \\ main \Mapsto^{\mathsf{prot}} \phi_{rel}(P_{\mathsf{True}}, P_{in}, P_{out}) * server \Mapsto^{\mathsf{prot}} \phi_{rel}(P_{lb\_v}, P_{in}, P_{out}) * server \mapsto v \end{array} \right\}$$
$$\langle n; \mathsf{serve\ main\ ip\ port\ server} \rangle$$
$$\{\mathsf{True}\}$$

This specification is fairly straightforward. It requires the `main` socket is bound to a socket protocol parameterized by P_{in} and P_{out} and the server to

be bound to the relay protocol with the ghost-resources described by P_{lb_v}. Also, `serve` expects an invariant that owns all resources needed to use the `main` socket. Moreover, the address matching the given ip address and port should not be in use already and the given port should be free on this ip address.

With these in place, the specification for the `load_balancer` module becomes:

$$\left\{\begin{array}{l} \overset{f}{\mapsto}(A) * (ip,p) \notin A * freePorts(ip,\{80\}) * IsNode(n) * \\ main \Mapsto^{\mathsf{prot}} \phi_{rel}(P_{\mathsf{True}}, P_{in}, P_{out}) * \\ \left(\ast_{s \in servers} \exists v.\, server \mapsto v * server \Mapsto^{\mathsf{prot}} \phi_{rel}(P_{lb\_v}, P_{in}, P_{out})\right) * \\ \left(\ast_{p \in [1100,\cdots,1100+length(servers)]} (ip,p) \notin A * freePorts(ip,\{p\})\right) \end{array}\right\}$$
$$\langle n; \mathsf{load\_balancer\ ip\ servers}\rangle$$
$$\{\mathsf{True}\}$$

Basically, the load balancer module require that the main socket protocol and all the server protocols are as described above. Furthermore for each server we should own a ghost-resource we could pass to `serve`. For each local port p in the range 1100 to $length(servers)$, which will become the endpoint for a corresponding server, p should be free and the address (ip, p) should not be in A.

In the Coq development accompanying Aneris we have shown an implementation of an addition service, both in the single case and in the load-balanced case.

5.6 Case Study 2a: Two-Phase Commit

A typical consensus problem in distributed systems is that of distributed commit; one operation should be performed by all participants in the system or none at all. A simplistic solution to distributed commit is the *one-phase* commit where a coordinator broadcasts to all participants to either commit or abandon a transaction. However, in distributed systems it is often the case that not all parties can commit due to other constraints - or a node could simply fail to comply by not receiving the message.

The *two-phase commit* protocol (TPC) due to Gray [27], is a distributed consensus protocol that coordinates all participants to either commit or abort. It is widely used in practice because it is somewhat resilient to a variety of failures, such as unreliable message delivery and transient participant crashes.

The reason for implementing and studying this protocol in Aneris is: (1) it is widely used in the real-world, (2) it is a complex network protocol thus serves as a decent benchmark for reasoning in Aneris, and (3) investigate if

the implementation can be given a specification that allow the TPC module to be used by a client that abstractly rely on some consensus protocol.

Implementation of the Two-Phase Commit Protocol

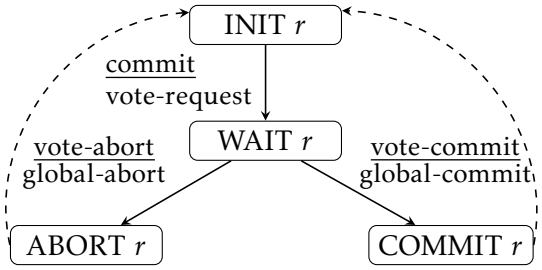
The two-phase commit protocol consists of the following two phases, each consisting of two steps:

1. a) The coordinator sends out a vote-request to each participant.
   b) A participant that receives a vote-request, replies with either vote-commit to inform that it is prepared to locally commit or a vote to abort.
2. a) The coordinator collects all votes and determines a result. If all participants voted commit, the coordinator sends a global-commit to all. Otherwise, at least one participant voted abort and the coordinator sends global-abort to all.
   b) All participants that voted for a commit wait for the final verdict from the coordinator. If the participant receives a global-commit it locally commits the transaction, otherwise the transaction is locally aborted. All must reply ACK.

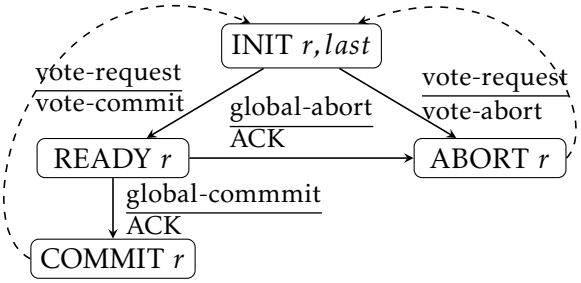
These steps are shown as finite state machines in Figure 59 and an implementation of a TPC-module that satisfies the conceptual description is shown in Figure 510. Our abstract model differs a bit from the standard diagram ([73]) because we reuse the same code and sockets for communication between coordinators and participants. Every state is therefore tagged with a unique round number and dashed arrows are local transitions allowing for reuse of the state-transition systems by incrementing round numbers. To allow each participant to locally transition to the INIT state upon round completion and still communicating commit or abort, the INIT state is tagged with the previous result (initially, COMMIT suffices).

The `tpc_coordinator` module expects an initial request message to be provided, along with a bound socket, a list of participants and a function to make a decision when all votes have been received. Internally, it uses two local references; one to collect all the votes and one to count the number of acknowledgments.

The `tpc_participant` module expects a socket and two handlers – one handler to decide on a vote and one handler to finalize on the decision made by the coordinator. When invoked, the module listens for incoming requests, decides on a vote and waits for a global decision from the coordinator. One could argue that `tpc_participant` is not faithfully implemented according to the TPC state-transition system because it always blocks until a decision is made by the coordinator. However, because each node can employ concurrency, the client can decide on concurrent work, in particular it can engage



(a) The coordinator state-transition system.



(b) The participant state-transition system.

Figure 59: The finite state machine for two-phase commit.

in other rounds of TPC with other coordinators. Notice as well that there are no round numbers in the implementation - these are purely in the abstract model to strengthen the specification.

Specification and Protocols

The approach for verifying programs in Aneris is similar to the approach in Iris; one has to consider the resources needed that correctly capture the intended purpose. We will use the following pre-created resources for a TPC instance, having a coordinator c and participants $p \in ps$:

- $parts(ps)$: Keeps track of all participants for a concrete TPC instance and is duplicable and unmodifiable. Conceptually, it fixes the participants in the TPC instance.
- $p \overset{c}{\mapsto} (r, CS)$: A participant-specific assertion on the state of the coordinator CS (as seen in Figure 59) for round r. The coordinator c owns an assertion regarding its own state $c \overset{c}{\mapsto} (r, CS)$. We require that $\exists r, CS. \forall a \in ps \cup \{c\}, a \overset{c}{\mapsto} (r, CS)$, that is all parties are in agreement of which round and state the coordinator has. Technically, this is stated as an invariant tpc_inv.

```
rec tpc_coordinate m skt ps dec :=
 let count := list_length ps in
 let msgs := ref (list_make ()) in
 let ack := ref 0 in
 list_iter (λ n := sendto skt m n) ps;
 listen skt (rec handler m from :=
  let msgs' := !msgs in
  msgs ← list_cons m msgs';
  if: list_length !msgs = count
  then () else listen skt handler);
  let res := dec !msgs in
  list_iter (λ n := sendto skt res n) ps;
  listen skt (rec handler m from :=
   ack ← !ack + 1;
   if !ack = count then res
   else listen skt handler)
```

```
rec tpc_participant skt vote fin :=
 let msg := listen_wait skt in
 let act := vote (fst msg) in
 sendto skt act (snd msg);
 let res := listen_wait skt in
 fin (fst res);
 sendto skt "ACK" (snd res);
 tpc_participant skt req fin

rec listen_wait skt :=
 match receivefrom skt with
   SOME m => m
 | NONE => listen_wait skt
 end
```

Figure 510: An implementation of the two-phase commit in AnerisLang. The functions `list_make`, `list_cons` and `list_length` are library utility functions for operations on lists implemented as splines.

- $p \overset{p}{\mapsto}\{\pi\}(r, PS)$: An assertion for each participant p regarding its own state. The resource can be split arbitrarily:

$$p \overset{p}{\mapsto}\{\pi\}(r, PS) \dashv\vdash p \overset{p}{\mapsto}\{\pi_1\}(r, PS) * p \overset{p}{\mapsto}\{\pi_2\}(r, PS)$$

  as long as $\pi_1 + \pi_2 = \pi \leq 1$ and $\pi_i > 0$. For any two fractions $p \overset{p}{\mapsto} \{\pi_1\} x * p \overset{p}{\mapsto}\{\pi_2\} x'$ we have $x = x'$. Finally, if one owns all fractions $p \overset{p}{\mapsto}\{1\}(r, PS)$, the resource can be freely updated since the environment cannot own any fractions.

To allow the client to use the TPC protocol in whatever fashion it wishes, the implementation is parameterized at all places dealing with messages. Consequently, when proving a client, the prover has to provide the decidable predicates *is_req*, *is_vote*, *is_abort* and *is_global* of type $(String \times \mathbb{N}) \to \mathrm{Prop}$. The client is free to pick $P : (Address \times String) \to iProp$ and $Q : (Address \times \mathbb{N}) \to iProp$, conceptually, the local pre- and post-condition for each participant. The socket

protocol for the coordinator is shown below.

$$\begin{aligned}
\phi_{vote} \triangleq \lambda m, \exists s, r, ps, sp. & \mathit{from}(m) = s * \mathit{parts}(\{s\} \cup ps) * \\
& \mathit{is_vote}(\mathit{body}(m), r) * s \overset{c}{\mapsto} (r, \mathsf{WAIT}) * \\
& (\mathit{is_abort}(\mathit{body}(m), r) * s \overset{p}{\mapsto}\{\tfrac{3}{4}\} (r, \mathsf{ABORT}) \vee \\
& \mathit{is_abort}(\mathit{body}(m), r) * s \overset{p}{\mapsto}\{\tfrac{3}{4}\} (r, \mathsf{COMMIT})) \\
\phi_{ack} \triangleq \lambda m, \exists m', cs, ps, pr, r, s. & s = \mathit{from}(m) * \mathit{parts}(\{s\} \cup ps) * \\
& s \overset{c}{\mapsto} (r, cs) * s \overset{p}{\mapsto}\{\tfrac{3}{4}\} (r, \mathsf{INIT}\ pr) * \\
& (cs = \mathsf{COMMIT} * pr = \mathsf{COMMIT} * Q(s, r) \vee \\
& cs = \mathsf{ABORT} * pr = \mathsf{ABORT} * P(m', s)) \\
\phi_{coord} \triangleq \lambda m, & \phi_{vote}(m) \vee \phi_{ack}(m)
\end{aligned}$$

For a participant p to send a vote to the coordinator c, it has to show that it is indeed a participant $\mathit{parts}(\{p\} \cup ps)$, that the message is a vote for that round and that it owns the resources for the state of c, $p \overset{c}{\mapsto} (r, \mathsf{WAIT})$, and a resource that confirms that the state of p matches p's vote. For the participant to send an acknowledgment, it has to prove it transitioned to INIT by sending $p \overset{p}{\mapsto}\{\frac{3}{4}\} (r, \mathsf{INIT}\ pr)$ where pr should match the global decision. If the decision was to commit, the participant should send the updated resources for Q, otherwise it should return the resources described by P. The socket protocol for the participant is as follows:

$$\begin{aligned}
\phi_{rec}(p) \triangleq \lambda m, \exists ps, r, sp. & \mathit{parts}(\{p\} \cup ps) * \mathit{is_req}(\mathit{body}(m), r+1) * \\
& \mathit{from}(m) \Mapsto^{\mathsf{prot}} \phi_{coord} * p \overset{c}{\mapsto} (r+1, \mathsf{WAIT}) * p \overset{p}{\mapsto}\{\tfrac{3}{4}\} (r, \mathsf{INIT}\ sp) \\
\phi_{glob}(p) \triangleq \lambda m, \exists ga, ms, ps, r, sc, sp. & \{\mathit{from}(m) | m \in ms\} = ps * \\
& \mathit{is_global}(\mathit{body}(m), r) * ga = \{m | m \in ms \wedge \mathit{is_abort}(m, r)\} * \\
& \mathit{parts}(ps) * \mathit{from}(m) \Mapsto^{\mathsf{prot}} \phi_{coord} * p \overset{c}{\mapsto} (r, sc) * p \overset{p}{\mapsto}\{\tfrac{3}{4}\} (r, sp) * \\
& \left(\mathop{\Large *}_{m \in ms} \exists m_{id}, \pi. \mathit{is_vote}(\mathit{body}(ms), r) * m_{id} \overset{m}{\mapsto}\{\pi\} m \right) * \\
& (ga = \emptyset \wedge \neg \mathit{is_abort}(\mathit{body}(m), r) \wedge sc = \mathsf{COMMIT} \vee \\
& (ga \neq \emptyset \wedge \mathit{is_abort}(\mathit{body}(m), r) \wedge sc = \mathsf{ABORT}) \\
\phi_{part}(p) \triangleq \lambda m, & \phi_{rec}(p)(m) \vee \phi_{glob}(p)(m)
\end{aligned}$$

In order to send a request for a round $r+1$ of TPC to a participant p, a coordinator c has to prove that request is valid and that the address the participant will reply on is bound to a coordinator protocol ($c \Mapsto^{\mathsf{prot}} \phi_{coord}$). Furthermore, the coordinator has to show it is in the WAIT state by transferring $p \overset{c}{\mapsto} (r+1, \mathsf{WAIT})$ and give up $p \overset{p}{\mapsto}\{\frac{3}{4}\} (r, \mathsf{INIT}\ sp)$ to allow the participant to make a transition.

Finally, the coordinator can broadcast a global decision to each participant when it has received all messages for a round. This is guaranteed by $(\ast_{m \in ms} \exists m_{id}, \pi. is\_vote(body(ms), r) * m_{id} \overset{m}{\mapsto} \{\pi\} m)$, where $\ast$ is iterated separating conjunction over finite sets. The coordinator also has to be honest in its decision, thus if any participant replied with an abort message, the global message should be abort as well and the final state of the coordinator should be ABORT. Also notice that for each message to a participant, the coordinator will pass in the assertion $(from(m) \Mapsto^{\mathsf{prot}} \phi_{coord})$. Therefore, the coordinator need not be primordial since the participant does not need to have prior knowledge of the coordinator. The coordinator could even change from round to round.

With the TPC protocol in place, we can finally give a specification to the two TPC modules. The tpc_participant specification is straightforward:

$$\left\{\begin{array}{l} tpc\_inv * parts(ps) * z_n \mapsto \mathsf{Some}\, p * p \Mapsto^{\mathsf{prot}} \phi_{part}(p) * p \overset{p}{\mapsto} \{\tfrac{1}{4}\}(r, \mathsf{INIT}\ sp) * \\ is\_reqf(req) * is\_finf(fin) \end{array}\right\}$$
$$\langle n; \mathsf{tpc\_participant\ z\ req\ fin} \rangle$$
$$\{\mathsf{True}\}$$

It requires ownership of a bound socket guarded by a participant protocol $\phi_{part}(p)$ and fractional ownership of its own state, initialized to be INIT. The specification for tpc_coordinate is a bit more involved:

$$\left\{\begin{array}{l} ps \equiv psV * is\_req(m, r+1) * tpc\_inv * parts(ps) * z_n \mapsto \mathsf{Some}\, a * \\ a \Mapsto^{\mathsf{prot}} \phi_{coord} * a \overset{c}{\mapsto} (r, \mathsf{INIT}) * is\_decf(dec) * \\ \ast_{p \in ps} \exists sp, p \Mapsto^{\mathsf{prot}} \phi_{part}(p) * p \overset{c}{\mapsto} (r, \mathsf{INIT}) * p \overset{p}{\mapsto} \{\tfrac{3}{4}\}(r, \mathsf{INIT}\ sp) * P(p, msg) \end{array}\right\}$$
$$\langle n; \mathsf{tpc\_coordinate\ m\ z\ psV\ dec} \rangle$$
$$\left\{\begin{array}{l} \langle n; v \rangle. \exists sc, sp. is\_global(v, r+1) * z_n \mapsto \mathsf{Some}\, a * a \overset{c}{\mapsto} (r+1, sc) * \\ \qquad \left(\ast_{p \in ps} p \overset{c}{\mapsto} (r, sc) * p \overset{p}{\mapsto} \{\tfrac{3}{4}\}(r, \mathsf{INIT}\ sp)\right) * \\ \qquad (is\_abort(v, r+1) * sc = \mathsf{ABORT} * sp = \mathsf{ABORT} * \ast_{p \in ps} \exists m. P(p, m) \vee \\ \qquad\quad \neg is\_abort(v, r+1) * sc = \mathsf{COMMIT} * sp = \mathsf{COMMIT} * \ast_{p \in ps} Q(p, r)) \end{array}\right\}$$

To call tpc_coordinate, one has to pass ownership of a socket z_n already bound to some address guarded by the ϕ_{coord} protocol. The list of nodes psV should be "equivalent" to the set of participants and for each participant the resources describing the participant's view of the coordinators and participant's state should be passed along. Additionally, we also need knowledge about the participant address being guarded by a suitable protocol $\phi_{part}(p)$.

The post-condition here is the most exciting part, mainly because it is exactly what one would expect. Either all participants along with the coordinator agreed to commit to which we obtain $Q(p, r)$ for each participant or they all agreed to abort, to which we get back $P(p, m)$. We elide the specifications *is_decf*, *is_reqf* and *is_finf* to the Coq development.

As to reap the fruits of our hard labor we show a client that use the TPC coordinator and participant modules.

5.7 Case Study 2b: Replicated Logging

As noted in Sergey et al. [70], clients of core consensus protocols have not received much focus from other major verification efforts ([28, 62, 81]) with the exception of ([50, 70]). In the work by Sergey et al. [70], both an implementation of TPC and a client for replicated logging with a side-channel for error recovery are verified. We will prove a similar client without a side-channel to allow for comparison with our verification efforts. The code is shown in Figure 511.

```
rec logger log addr m dbs :=
  let skt := socket() in
  let dec := λ msgs :=
    let r = list_fold (λ a m :=
      a && m = "COMMIT") true msgs in
    if r then "COMMIT" else "ABORT" in
  socketbind skt addr;
  tpc_coordinate ("REQUEST_" ^ m)
                 skt dbs dec
```

```
rec db addr :=
  let skt := socket() in
  let wait := ref "" in
  let log := ref "" in
  let req := λ m := wait ← val_of m;
                    "COMMIT" in
  let fin := λ m := if m = "COMMIT"
             then log ← !log ^ !wait
             else () in
  socketbind skt addr;
  tpc_participant skt req fin
```

Figure 511: Replicated logging that use two-phase commit modules in Aneris-Lang. ^ is string concatenation.

The replicated `logger` opens a socket `skt` on address `addr` and initiates a TPC-round for all databases `dbs` by sending `"REQUEST_" ^ msg`. The decision handler `dec` is called by the TPC coordinator module when all votes have been received.

On the side of the database, `db`, we use an internal reference `log` as the log [6]. Upon an incoming request, the message is parsed (`val_of m`) and the log to append is stored in the reference `wait`. If the global decision by `logger` is "COMMIT", the string stored in `wait` will be appended to the `log`.

To logically describe the local state of each database we use the following heap-like predicates, $p \overset{l}{\mapsto} \{\pi\}\, log$ and $p \overset{w}{\mapsto} \{\pi\}\, log, wait$, that keep track of the

[6] Ideally, this should be "persistent" storage, however, to keep the example concrete, we use a local reference

log and waiting commit for each participant p. The predicate P and Q, which we instantiate TPC with are defined below:

$$P \triangleq \lambda p, m. \exists log, s.\ m = "REQUEST\_" @ s * p \overset{l}{\mapsto} \{\tfrac{1}{2}\} log * p \overset{w}{\mapsto} \{\tfrac{1}{4}\} log, s$$

$$Q \triangleq \lambda p, n. \exists log, s.\ p \overset{l}{\mapsto} \{\tfrac{1}{2}\} log @ s * p \overset{w}{\mapsto} \{\tfrac{1}{4}\} log, s$$

With these resources in place, we can give the `logger` the following specification:

$$\left\{ \begin{array}{l} tpc\_inv * parts(dbs) * freePorts(ip(addr), \{port(addr)\}) * is\_req(m) \\ \ast_{p \in dbs} \exists sp, p \Rightarrow^{\mathsf{prot}} \phi_{part}(p) * p \overset{c}{\mapsto} (r, \mathsf{INIT}) * p \overset{p}{\mapsto} \{\tfrac{3}{4}\} (r, \mathsf{INIT}\ sp) * P(p, m) \end{array} \right\}$$

$$\langle n; logger\ log\ addr\ dbs \rangle$$

$$\left\{ \begin{array}{l} \langle n; v \rangle. \exists m, r. \ast_{p \in dbs} \exists sp.\ p \overset{c}{\mapsto} (r, \mathsf{INIT}) * p \overset{p}{\mapsto} \{\tfrac{3}{4}\} (r, \mathsf{INIT}\ sp) * \\ \qquad \left(v = "COMMIT" * \ast_{p \in dbs} Q(p, r) \vee v = "ABORT" * \ast_{p \in dbs} P(p, m) \right) \end{array} \right\}$$

Verification of our replicated logging client using two-phased-commit follows directly, in a modular node-local fashion, from applying the specification of `tpc_coordinate`.

We assume that the consensus-protocol could be swapped with other consensus protocols, such as strict versions of Raft or Paxos, by making the protocol parametric in the code and proof. Raft and Paxos require only a majority of participants for committing, which is observationally different than two-phase commit, thus strict version of these consensus protocols would be required to align functionality with two-phase commit.

Interestingly, different clients can easily be built upon TPC because the two-phase commit module is parametric in the shape of messages sent. This could be an auction service where each request carries an item number and each commit response is a bid, or an election service where each request is a list of candidates and each response is a vote.

5.8 Related Work

Verification of distributed systems has received a fair amount of attention. In order to give a better overview, we have divided related work into four categories.

Model-Checking of Distributed Protocols

Previous work on verification of distributed systems has traditionally focused on verification of protocols or core network components by means of model-

checking. Frameworks for showing safety and liveness properties, such as SPIN Holzmann [30], and TLA+ [46], have had great success. A clear benefit of using model-checking frameworks is that they allow to state both safety and liveness assertions as LTL assertions [59]. Mace [37] provides a suite for building and model-checking distributed systems with asynchronous protocols, including liveness conditions. Chapar [50] allows for model-checking of programs that use causally consistent distributed key-value stores. Neither of these languages provide higher-order functions or thread-based concurrency. Additionally, model-checking frameworks cannot prove the absence of errors in general, they can only show it for a specific model.

Session Types for Giving Types to Protocols

Session types have been studied for a wide range of process calculi, in particular, typed π-calculus. The original idea was to describe two-party communication protocols as a type to ensure communication safety and progress [31]. This was later extended to multi-party asynchronous channels [32], multi-role types [19] which informally model topics of actor-based message-passing and dependent session types allowing quantification over messages [77]. Our socket protocol definitions are quite similar to the multi-party asynchronous session types in the sense that our sockets are multi-party as well and progress can be encoded by having suitable ghost-assertions and using magic-wand (§5.2).

Distributed Hoare Type Theory and Concurrent Program Logics

Disel [70] is a Hoare Type Theory for distributed program verification in Coq with ideas from separation logic. It provides the novel protocol-tailored rules WithInv and Frame which allow for modularity of proofs under the condition of an inductive invariant and distributed systems composition. We obtain composition by node-local reasoning, which is possible because we work in a logic that requires us to state stable assertions on the environment locally.

Disel cannot hide mutable state in the system and it must be made known to all in a protocol. Additionally, node-local mutable state can only be altered upon send/receive. Besides having thread-based concurrency on nodes, AnerisLang also provides "proper" local mutable state that can be hidden, which the authors of Disel described as interesting future work.

However, in Disel, programs can be extracted into runnable OCaml programs, which is on our agenda for future work.

IronFleet [28] allows for building provably correct distributed systems by combining TLA-style state-machine refinement with Hoare-logic verification in a layered approach, all embedded in Dafny [49]. IronFleet even allows for liveness assertions. The top layer is a simple specification of the system's behavior, the bottom layer is the actual implementation, and each layer is

proven to satisfy the layer above it. The verification results of IronFleet are impressive, however, it seems like composition of protocols requires progressing changes through the entire stack of layers. Compared to IronFleet, verified components in Aneris are easier to compose and can employ thread-local reasoning.

The concurrent program logic closest to our work is naturally Krebbers et al. [38], since it is the foundation we have based Aneris upon. Other concurrent program logics, [20, 55, 56, 72, 80] to name a few, all have some notion of thread-local reasoning or rely-guarantee reasoning. We believe we have successfully lifted that principle to reason about individual nodes locally in distributed systems.

Other Distributed Verification Efforts

Verdi [81] is a framework for writing and verifying implementations of distributed algorithms in Coq, providing a novel approach to network semantics and fault models. To achieve compositionality, the authors introduced *verified system transformers*, that is, a function that transforms one implementation to another implementation, which has different assumptions about its environment. This makes vertical composition difficult for clients of proven protocols and in comparison, AnerisLang feature set is more expressive.

EventML [62, 63] is a functional language in the ML family that can be used for coding distributed protocols using high-level combinators from the Logic of Events, and verify them in the Nuprl interactive theorem prover. It is not quite clear how modular reasoning works, since one works within the model, however, the notion of a central main observer is akin to our system node.

5.9 Conclusion and Future Work

Distributed systems are quite ubiquitous nowadays and hence it is essential to be able to verify them. In this paper we presented Aneris, a framework for writing and verifying distributed systems in Coq on top of the framework of the Iris program logic. From a programming point of view, the important aspect of Aneris is that it is quite feature rich: it is basically a concurrent ML-like programming language with network primitives. This allows individual nodes to internally use higher-order heap and concurrency to write efficient programs.

On the program logic side the Aneris framework provides node-local reasoning. That is, we can reason about individual nodes in isolation as we reason about individual threads in isolation in what we refer to as node-local reasoning. We demonstrated the versatility of Aneris by studying interesting distributed systems both implemented and verified within Aneris. The adequacy theorem of Aneris implies that these programs are safe to run.

Module	Impl	Spec	Proofs
Load Balancer (§5.5)			
Load-balancer	18	78	95
Addition Service			
Server	11	15	38
Client	9	14	26
Adequacy (1 server, 2 clients)	5	12	62
Adequacy w. Load Balancing (3 servers, 2 clients)	16	28	175
Two-phase commit (§5.6)			
Coordinator	18	181	265
Participant	11		280
Replicated logging (§5.7)			
Instantiation of TPC	-	85	-
Logger	22	19	95
Database	24	20	190
Adequacy (2 dbs, 1 coordinator, 2 clients)	13	-	137

Table 51: Sizes of implementations, specifications and proofs in lines of code for modules. The addition service is a simple server that listens for incoming messages consisting of two numbers and responds with the sum. When proving adequacy the system must be closed.

Relating the verification sizes of the modules from Table 51 to other formal verification efforts in Coq indicates that it is easier to specify and verify systems in Aneris. The total work required to prove two-phase commit with replicated logging is 1,272 lines which is just halve of the lines needed for proving the inductive invariant for TPC in [70]. However, extensible work has gone into the Iris Proof-mode thus it is hard to conclude that Aneris requires less verification effort and not just have richer tactics.

As of writing, AnerisLang is suitable for verifying code written in other operationally similar languages, such as OCaml, by writing it in the DSL defined in Coq. However, because the language is so realistic, one could write a simple transpiler for a *one-to-one* translation of terms in AnerisLang to terms in OCaml, without the need for code-extraction. A transpiler would make it feasible to use AnerisLang for a verify-first approach.

In fact, for an earlier version of AnerisLang, we had an existing compiler with a small standard library consisting of 15 lines of OCaml, mainly for parsing received bytes from sockets to strings.

Chapter 6

Verifying a Concurrent Data-Structure from the Dartino Framework

MORTEN KROGH-JESPERSEN, Aarhus University, Denmark
THOMAS DINSDALE-YOUNG, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

The formal development accompanying this research project can be found on the Iris project web-site https://iris-project.org. As of writing a direct link for the development is https://iris-project.org/artifacts/2017-case-study-verifying-dartino-framework.zip.

Abstract

We specify and verify a concurrent queue data structure used in the scheduler of a real-world virtual machine, Google's Dartino Framework.

Our specification treats the queue operations as abstractly atomic. This means that a client can reason about them as if they take effect at a single instant in time, and thus impose its own invariants on the queue. The specifications also involve resource transfer: to enqueue a process, a thread transfers ownership of its descriptor to the queue.

We show that an implementation of the data structure, directly translated from the Dartino Framework source, satisfies our specification in Iris, a state-of-the-art higher-order concurrent separation logic, capable of expressing both abstract atomicity and resource transfer. Our verification is formalised in the Coq proof assistant. Hence, our work shows that Iris is both expressive and practical enough to formally reason about production code taken from "the wild".

6.1 Introduction

The scheduler is the beating heart of any virtual machine – it is responsible for running and pausing processes of the system. Therefore, the scheduler must be both correct and efficient. The Dartino Framework is a virtual machine for the Dart language, which was designed by Google to run efficiently on limited hardware (such as embedded systems or IoT-devices). The work presented here is the result of a collaboration with the Google Dartino team to verify the queue data structure underlying the Dartino Framework's scheduler.

The Dartino Framework uses a pool of low-level (hardware) *threads* to run high-level Dart *processes*. Each thread has its own process queue, implemented as a doubly-linked list which we refer to as a *Dartino Queue*. Having a queue per thread serves to reduce contention, although threads may access the queues of other threads. For instance, a thread with no processes may steal one from another thread. In addition to the usual enqueue and dequeue operations, the data structure allows a specific process to be removed from anywhere in the queue. This allows the scheduler to prioritise certain processes – for instance, to immediately schedule a process that is the recipient of a message.

Verification of sequential implementations of doubly-linked lists using shape analysis or separation logics has already been studied in detail, *e.g.* in the seminal work by Reynolds [57, 66]. Specifying and verifying the Dartino Queue is complicated by a number of factors.

Firstly, this Dartino Queue allows for concurrent access by multiple threads. We therefore require a specification that accounts for this. *Abstract atomicity* achieves this by specifying that an operation (such as enqueuing a process) appears to take effect at a single instant in time. A client can then reason about abstractly atomic operations in a simple manner, for instance by imposing new invariants on how the queue is used. Linearizability [29] is a well-known verification condition for abstract atomicity. Recently, notions of abstract atomicity have been introduced to separation logics such as TaDA [18].

Secondly, during its lifetime, a process may belong to multiple queues. This means that ownership of a process descriptor is transferred whenever it is enqueued or dequeued. This ownership transfer does not necessarily take place at the same instant that the operation atomically takes effect. Separation logics are well-equipped to reason about resource transfer; consequently, a separation logic which supports abstract atomicity is appropriate for this verification problem.

We have chosen to verify the Dartino Queue in Iris [33, 38], a state-of-the-art concurrent higher-order separation logic, implemented in the Coq proof assistant [17]. The reason for this is that the Iris Proof Mode enables us to do interactive proofs directly in Coq [39] and, moreover, Iris allows us to prove so-called atomic triples [18], which capture abstract atomicity. We

can therefore give strong specifications that integrate ownership transfer and abstract atomicity.

Our case study applies Iris to verifying real-world code with non-trivial specifications. Our case study demonstrates the practicality and effectiveness of the following:

- Using resources in Iris to reason about dynamic allocation and stealing of processes which may be transferred between queues.
- Using logical atomicity in Iris in concert with resource transfer to verify strong specifications that accurately capture the intention for the real-world code.
- Using the Iris Proof Mode for formal, mechanised verification of code.

Outline. First, we describe the Dartino Queue and show the translation from C++ to Iris in §6.2. In §6.3 we give a primer to Iris and describe the invariants that will guard the Dartino Queue in §6.4. In §6.5 we motivate and show stronger specifications for the operations on the Dartino Queue before showing a client of the queue in §6.6. Finally, we conclude in §6.7.

6.2 The Dartino Queue in Iris

The Dartino Framework is an experimental virtual machine, written in C++, for running the programming language Dart on devices with limited memory and limited processing resources. One particular goal with the Dartino Framework is to increase the computation throughput of concurrent programs that use message passing for communication. To this end, when one Dart process sends a message to another, the recipient is preferentially scheduled. This means that the Dartino Queue, which represents a process queue in the scheduler, must allow for processes that are not at the head to be removed from the queue.

In a general-purpose queue data structure, enqueuing a value typically involves allocating a new node to hold the value. For a process queue, however, the process descriptor, which exists for the lifetime of the process, directly represents a node in a queue. That is, the descriptor object holds pointers to the queue the process belongs to and its adjacent processes. This means that no allocation is necessary in enqueuing a process (which is good, since allocation is expensive and the scheduler must be as efficient as possible). On the other hand, one must handle ownership of process objects carefully, since they may belong to multiple queues during their lifetimes.

The Dartino Queue is implemented as a doubly-linked list to support removal of an arbitrary process in its queue. Updating a doubly-linked list requires multiple pointer updates. To ensure that these updates occur safely

```
Definition unSOME :=  λ: p, match: p with NONE => assert false
                                        | SOME p' => p' end.

Definition queue_head := λ: p, Fst p.
Definition queue_tail := λ: p, Fst (Snd p).
Definition queue_sent := λ: p, Snd (Snd p).

Definition pval := λ: p, Fst p.
Definition qref := λ: p, Fst (Snd p).
Definition prev := λ: p, Fst (Snd (Snd p)).
Definition next := λ: p, Snd (Snd (Snd p)).

Definition makeQueue := λ: <>, (ref NONE, (ref NONE, ref ())).

Definition makeProc :=
  λ: v, (ref v, (ref NONE, (ref NONE, ref NONE))).

Definition obtainLockDeq :=
  rec: loop head sentinel h :=
    let: hv := !h in
    if: (hv = SOME sentinel) || (~ CAS head hv (SOME sentinel))
    then h <- !head ;;
         if: (!h) = NONE then true
         else loop head sentinel h
    else false.

Definition dequeue :=
  λ: head tail s,
     λ: <>,
        let: h := ref !head in
        if: !h = NONE then NONE
        else let: obtLock := obtainLockDeq head s h in
             if: obtLock then NONE
             else let: h' := unSOME (!h) in
                  let: next := !(next h') in
                  (if: next = NONE then tail <- NONE
                   else prev (unSOME next) <- NONE);;
                        next h' <- NONE;;
                        qref h' <- NONE;;
                        head <- next;;
                        SOME h'.
```

Figure 61

```
Definition obtainLockEnq :=
  rec: loop head sentinel h :=
    let: hv := !h in
    if: (hv = SOME sentinel) || (~ CAS head hv (SOME sentinel))
    then h <- !head ;; loop head sentinel h
    else ().

Definition enqueue :=
  λ: head tail s,
     λ: p, let: h := ref !head in
           obtainLockEnq head s h;;
           qref p <- SOME (head,(tail,s));;
           match: !h with
             NONE => tail <- SOME p;;
                     head <- SOME p;;
                     true
           | SOME h' => prev p <- !tail;;
                        next (unSOME !tail) <- SOME p;;
                        tail <- SOME p;;
                        head <- SOME h';;
                        false
            end.

Definition tryDequeueEntry :=
  λ: head tail s,
     λ: p, let: h := ref !head in
           if: !h = NONE then false
           else let: obtLock := obtainLockDeq head s h in
               if: obtLock then false
               else if: !(qref p) = SOME (head,(tail,s))
                    then let: next := !(next p) in
                         let: prev := !(prev p) in
                         (if: next = NONE
                         then tail <- prev
                         else prev (unSOME next) <- prev);;
                              (if: prev = NONE
                              then h <- next
                              else next (unSOME prev) <- next);;
                              (prev p) <- NONE;;
                              (next p) <- NONE;;
                              (qref p) <- NONE;;
                              head <- !h;;
                              true
                    else head <- !h;;
                         false.
```

Figure 61: Implementation of a doubly-linked queue with a virtual lock. Function binders in Iris-ML are strings in Coq, but are shown as regular binders for the sake of clarity.

in a concurrent context, the Dartino Queue uses the queue's head pointer as a spin lock.

Modelling C++ in Iris-ML

In order to verify the Dartino Queue, we translate the C++ code used by Google into Iris-ML, one of the programming languages supported by Iris. In doing so, we must faithfully represent the semantics of the original program. In particular, memory operations should have the same granularity: the ML program cannot perform an update in a single atomic step that takes multiple steps in the C++ source.

In C++, an object is represented as a contiguous block of memory holding the object's data members. A pointer to an object is the address of such a block, and members are accessed by computing offsets from the address into the block.

In Iris-ML, there are no objects, but there are references to arbitrary (untyped) values. The basic operations on references are:

- `ref v` — allocate a reference with initial value `v`;
- `!r` — atomically read the value stored in reference `r`;
- `r <- v` — atomically update the contents of reference `r` to value `v`; and
- `CAS r oldval newval` — atomically compare the contents of reference `r` with value `oldval`, updating it to `newval` if equal; return `true` if successful (the value was updated) and `false` otherwise.

One way a C++ object reference might be represented in Iris-ML is as a reference to a tuple of the object's data members. This representation is problematic, however, since any update to the object updates all of its members at once, while in C++ each data member is updated individually. Consequently, a C++ object reference is represented as a tuple of references to each of the object's data members. Each data member can thus be manipulated independently.

Apart from a reference to an object, a C++ pointer may instead hold the value `null`. To reflect that pointers are nullable in Iris-ML, we represent pointers as tagged data: `NONE` represents the null pointer, and `SOME r` represents a pointer with a valid object reference `r`. To dereference a pointer, we first apply the function `unSOME`, which strips the `SOME` tag and crashes when given `NONE`.

Doubly-Linked List with Arbitrary Removal

The interface of the Dartino Queue consists of five operations:

`makeQueue`: Construct a new Dartino Queue.

`makeProc`: Construct a new process descriptor.

`enqueue`: Append a process to a Dartino Queue.

`dequeue`: Attempt to remove the first process from a Dartino Queue, returning a pointer to the process. This can fail, returning a `null` pointer (Iris-ML: `NONE`), if the queue is empty.

`tryDequeueEntry`: Attempt to remove a specified process from a Dartino Queue. This can fail, returning `false`, if the process is no longer in the queue.

The Iris-ML implementation is given in Figure 61. We now describe each operation in detail.

New Dartino Queue. The function `mkQueue()` creates a new, empty Dartino Queue. A Dartino Queue object has three data members: the pointers `head` and `tail` to the head and tail of the queue, and a distinguished sentinel value `sent`. To indicate when the lock on the queue is held, the `head` pointer is set to the sentinel. To ensure that this sentinel value is distinct from any process reference, the `makeQueue` constructor generates a new reference (whose contents is immaterial). The `head` and `tail` pointers are both initialised to `NONE` (representing `null`). Figure 62 shows the initial configuration of a Dartino Queue object.

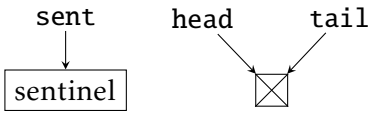
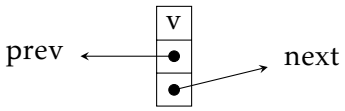


Figure 62: Initial configuration of the queue

New Process. The function `mkProcess(v)` constructs a new process descriptor holding value `v`. (The meaning of the value is determined by the client of the queue, which in the Dartino Framework is the process's instruction pointer.) A process descriptor has four members: a value `pval`; a pointer to the queue that currently holds the process, `qref`; and pointers `prev` and `next` to the previous and next processes in the queue, respectively. We depict processes as so (where the `qref` pointer is not drawn since the queue that owns the process is obvious from the context):



In Iris-ML, a process object is represented as a tuple of references, and we define four projections out of the tuple named `pval`, `qref`, `prev` and `next`.

Enqueuing. The function `enqueue(q,p)` enqueues process `p` in the Dartino Queue `q`. Enqueuing elements involves obtaining the (virtual) lock of the Dartino Queue, inserting the new element once the lock is acquired, and finally releasing the lock again. These steps are illustrated in Figure 63.

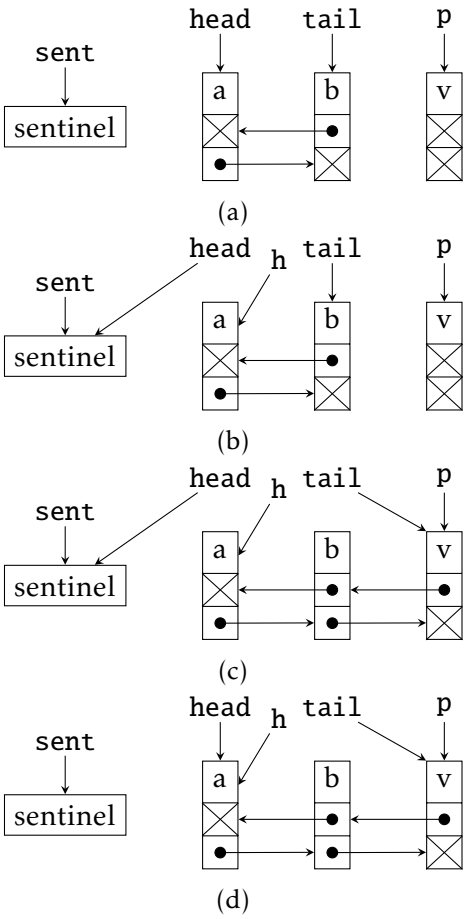


Figure 63: Enqueuing an element into the Dartino Queue.

Obtaining the lock is delegated to `obtainLockEnq`, which loops attempting to update the head pointer of the queue (`head`) to the sentinel value (`sentinel`); the old value of the head pointer is recorded in the reference `h`. The function retries if the head currently holds the sentinel value (indicating that another thread holds the lock) or if the `CAS` fails as a result of another thread updating it. When `obtainLockEnq` returns, it must have successfully updated the head pointer from the (non-sentinel) value now stored in `h` to the

sentinel value. Thus the thread will have acquired the lock. Obtaining the lock takes us from to a in Figure 63.

Once the lock is held, the thread is at liberty to modify the list, and can assume that no other thread will concurrently modify it. The process is added to the end of the list by performing four pointer updates: the process's `qref` pointer is updated to point to the queue; the process's `prev` pointer is updated to point to the original `tail`; the tail's `next` pointer is updated to point to the new process; and the `tail` pointer is updated to point to the new process. This update takes us from a to b in Figure 63. In the case where the list was initially empty, it is only necessary to update the process's `qref` pointer and the queue's `tail` pointer.

To complete the enqueue operation, the `head` pointer is updated to point to the original head of the list (which was stored in `h`), if the list was non-empty. This takes us from b to c in Figure 63. If the list was empty, the `head` pointer is updated to point to the newly enqueued process.

Dequeuing. The function `dequeue(q)` dequeues the process at the head of the Dartino Queue `q`. As with enqueuing, the operation involves acquiring the lock, updating the list, and finally releasing the lock. This is depicted in Figure 64.

Before attempting to obtain the lock, a test checks if the head pointer was `NONE`, indicating that the queue was empty, in which case the function immediately returns `NONE`. Otherwise, an attempt to acquire the lock is made by calling `obtainLockDeq`.

`obtainLockDeq` behaves like `obtainLockEnq` in acquiring the lock, except that it does not attempt to acquire the lock if the queue is empty; it returns `true` if the queue was empty, and therefore the lock was not acquired, and `false` if the lock was successfully acquired with the queue non-empty.

When the lock is successfully acquired, `h` holds a (non-null) pointer to the process descriptor at the head of the queue (Figure 64 a). The descriptor's `next` pointer is inspected to determine if it is the end of the queue, in which case it will be `NONE`. If so, the queue's `tail` pointer is set to `NONE` since the queue will now be empty. If not, the next process's `prev` pointer is set to `NONE`, since it will now be the head of the queue. The `next` and `qref` fields of the head process are both updated to `NONE`, since it is being removed from the queue (Figure 64 b). Finally, the queue's `head` is updated to point to the new head process (the successor of the removed process, before it was removed).

Arbitrary Dequeuing. The most interesting aspect of the Dartino Queue is that a specific process `p` can be removed from a queue `q` with the function `tryDequeueEntry(q,p)`. This is shown in Figure 65.

As with `dequeue`, the first step is to acquire the lock for the queue, but only if the queue is non-empty. If the queue is empty then the process

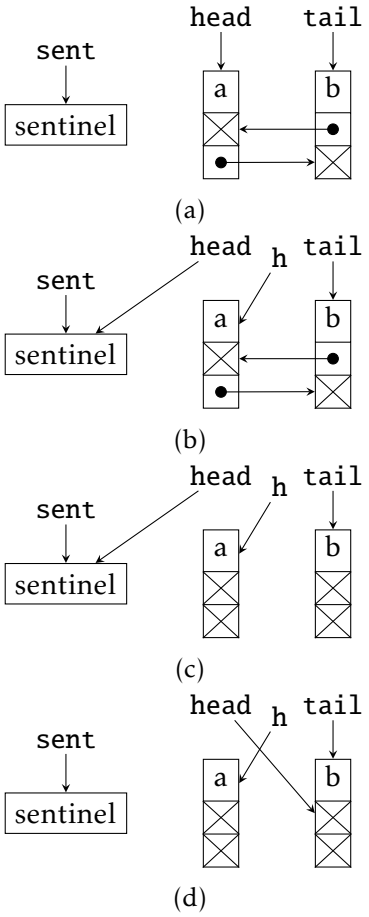


Figure 64: Dequeuing an element from the Dartino Queue.

cannot be in the queue (perhaps another thread already dequeued it) and so `tryDequeueEntry` returns `false`. Otherwise, it is necessary to check that the process's `qref` pointer points to the queue, since even if this was initially the case, another thread may have dequeued the process since `tryDequeueEntry` was called. If `qref` does not match the queue, the lock is released by updating `head` to its previous value and the operation returns `false`. Otherwise, we can be sure that the process indeed belongs to the queue, and since the thread holds the lock on the queue, no other thread can concurrently dequeue it (Figure 65 a).

The process `p` is removed from the queue by first updating the `prev` pointer of its successor to the `prev` pointer of `p`. If the successor is `NONE` then the `tail` pointer is updated instead, since `p` must be the last process in the queue. Next, the `next` pointer of `p`'s predecessor is updated to point to the `next` pointer of `p`. Again, if there is no predecessor, the `h` pointer is updated instead, since `p`

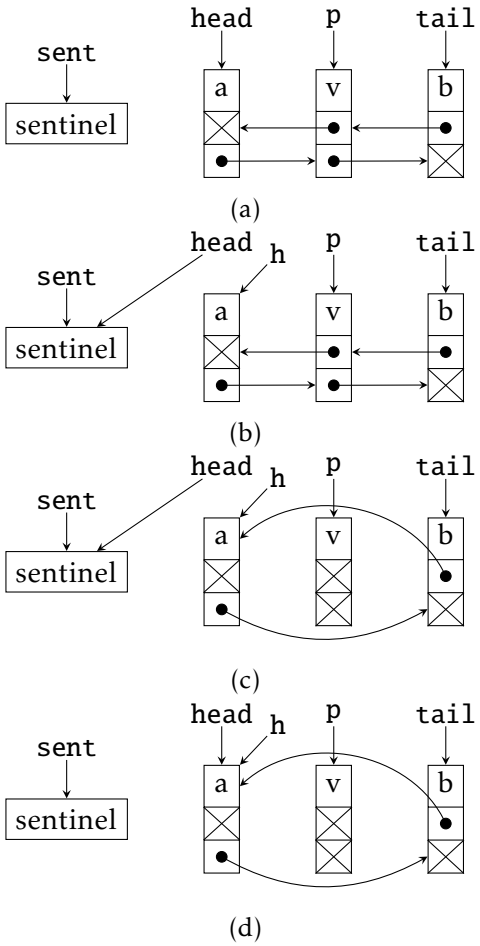


Figure 65: Dequeuing a specific element `p` from the Dartino Queue.

must be the first process in the queue. Next, the `prev`, `next` and `qref` pointers of `p` are all set to `NONE` (Figure 65 b). Finally, the lock on the queue is released by updating `head` to the pointer stored in `h`, which is either the former head of the queue (if it was not `p`) or the successor of `p` (if `p` used to be the head).

6.3 The Iris Logic

We specify and verify the Dartino Queue in Iris, a concurrent higher-order separation logic implemented in Coq. Iris is built around monoids and invariants. Monoids provide a way to define abstract (ghost) resources that represent knowledge and rights available to threads. Invariants provide a way to give concrete meaning to these abstract resources.

Iris includes the following quantifiable types:

$$\kappa ::= \mathbf{1} \mid \kappa \times \kappa \mid \kappa \to \kappa \mid \mathit{Expr} \mid \mathit{Val} \mid \mathbb{B} \mid \mathbb{N} \mid \mathit{Names} \mid \mathit{Monoid} \mid \mathit{iProp} \mid \ldots$$

Here, $\mathbf{1}$, $\mathbb{B}$ and $\mathbb{N}$ is the unit type, the type of booleans and the type of natural numbers respectively. *Expr* and *Val* are syntactic expressions and values of Iris-ML. *Monoid* is the type of monoids, which are used for ghost resources. *Names* is the type of ghost names, which is used to assign names to instances of ghost resources. *iProp* is the type of Iris propositions, which are defined by the following grammar:

$$\begin{aligned} P ::= {} & \top \mid \bot \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid P * P \mid P \mathrel{-\!\!*} P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \\ & \mid \triangleright P \mid \mu r.P \mid \checkmark(a) \mid \Box P \mid \models\{\mathtt{E1},\mathtt{E2}\}\Rightarrow P \mid \mathtt{own}\ \gamma\ \mathtt{a} \mid \mathtt{inv\ N\ P} \mid \ldots \end{aligned}$$

The grammar includes the usual higher-order logic connectives ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$). The separating conjunction, $*$, describes resources that are split into two disjoint parts. Magic wand, $P \mathrel{-\!\!*} Q$, behaves like implication for resources: if resources P are given up, then the resources described by Q can be obtained. Ownership of ghost resources is written as $\mathtt{own}\ \gamma\ \mathtt{a}$ where $\mathtt{a}$ is a monoid element and γ is a ghost name. Owned resources must be valid, which is asserted by $\checkmark(\mathrm{a})$. The update modality $\models\{\mathtt{E1},\mathtt{E2}\}\Rightarrow P$ allows resources to be updated to resources satisfying P, where the masks $\mathtt{E1}$ and $\mathtt{E2}$ describe the set of invariants that are closed before and after the update, respectively. When the masks are the same we write $\models\{\mathtt{E}\}\Rightarrow P$.

The $\Box$ modality asserts that a proposition holds independently of resources. Consequently, the proposition $\Box P$ is *persistent*: it can be freely duplicated as it satisfies $\Box P \Leftrightarrow \Box P * \Box P$.

Invariants, $\mathtt{inv\ N\ P}$, where $\mathtt{N}$ is the name of the invariant and $\mathtt{P}$ the invariant assertion, are also persistent propositions and hence duplicable. The resources embodied by an invariant can only be accessed during atomic operations, which must reestablish the invariant. This ensures that no thread can see a violation of the invariant: it is indeed invariant.

Updating resources and opening invariants modify the ghost state without executing any code. The following two connectives, *view shift* and *wand shift*, are particularly useful for these tasks:

$$\begin{aligned} P =\{\mathtt{E1},\mathtt{E2}\}\Rightarrow Q &\triangleq \Box(P \mathrel{-\!\!*} \models\{\mathtt{E1},\mathtt{E2}\}\Rightarrow Q) \\ P =\{\mathtt{E1},\mathtt{E2}\}=* Q &\triangleq P \mathrel{-\!\!*} \models\{\mathtt{E1},\mathtt{E2}\}\Rightarrow Q \end{aligned}$$

The behaviour of view shifts and wand shifts are similar to Hoare-triples, taking a precondition P, that asserts the shape of the ghost state before updating and a postcondition Q, the ghost state obtained by running the view shift or wand shift. There is not need for any code, since these shifts only

$$\textsc{Monoid-Alloc}\ \frac{\checkmark(a)}{\models\{E\}\Rightarrow\exists\gamma,\ \mathsf{own}\ \gamma\ a} \qquad \textsc{Monoid-Update}\ \frac{a \rightsquigarrow B}{\mathsf{own}\ \gamma\ a \vdash \models\{E\}\Rightarrow\exists b \in B,\ \mathsf{own}\ \gamma\ b} \qquad \textsc{Monoid-Valid}\ \ \mathsf{own}\ \gamma\ a \vdash \checkmark(a)$$

$$\textsc{Monoid-Op}\ \ \mathsf{own}\ \gamma\ a * \mathsf{own}\ \gamma\ b \dashv\vdash \mathsf{own}\ \gamma\ a \cdot b$$

Figure 66: Rules for monoid resources in Iris

update ghost resources and not the physical state. Note that the view shift is persistent: it cannot depend on any currently-available resources. By contrast, the wand shift may use up available resources in the update.

Monoids

Commutative monoids are the bread and butter of any separation logic. A commutative monoid consists of a set with a binary operation (which we denote $\cdot$) that is associative, commutative and has an identity element. In Iris, arbitrary monoids can be used as ghost resources. (Technically, Iris uses *resource algebras*, which relax the identity property, but have some other properties. We will abuse the terminology by referring to resource algebras as monoids even when they do not have an identity element.)

Figure 66 shows Iris rules for working with monoid resources. The Monoid-Alloc rule allows a new ghost resource to be allocated, holding any valid monoid element. The Monoid-Valid rule requires that any allocated resource must hold a valid monoid element. The Monoid-Update rule allows a ghost resource to be updated. The $\rightsquigarrow$ represents frame-preserving update: if $a \rightsquigarrow B$ and $\checkmark(a \cdot c)$ then it must be that $\checkmark(b \cdot c)$ holds for some $b \in B$. Frame-preserving updates thus do not invalidate the ownership of any other concurrent threads. The Monoid-Op rule allows ghost resources to be split and joined with the monoid composition operator.

We now present a number of standard monoid constructions that are useful in our verification efforts.

Exclusive. The $\mathrm{Ex}(S)$ monoid (over a given set S) is one of the simplest monoids, where the composition $a \cdot b$ is undefined everywhere. ("Undefined" is represented by a distinguished element of the monoid that is not valid.) For the exclusive monoid, we thus have the following law:

```
∀ a b, own γ (Excl a) ∗ own γ (Excl b) ⊢ ⊥.
```

There cannot be two owned instances at one point in time, therefore one is always free to update the resource using the Monoid-Update rule.

Decidable Agreement. The DecAgree(S) monoid over a set S with decidable equality has composition defined by $s \cdot s = s$ for all $s \in S$, but undefined otherwise. This means that if two threads own elements of this monoid then they must agree. This is expressed by the following Iris proposition:

```
∀ a b, own γ (DecAgree a) ∗ own γ (DecAgree b) ⊢ a = b.
```

To show this holds, we use Monoid-Op to combine the components into one assertion. Then, by Monoid-Valid, we have that $a \cdot b$ is valid, but that can only be true if $a = b$. Notice that we cannot perform a frame-preserving update for this monoid.

Fractional Permissions. Given a monoid M, the fractional permissions monoid Frac(M) has carrier $(\mathbb{Q} \cap (0,1]) \times M$. Composition is defined as $(\pi_1, a) \cdot (\pi_2, b) = (\pi_1 + \pi_2, a \cdot b)$, where the sum $\pi_1 + \pi_2$ may not exceed 1. We thus have the following Iris propositions:

```
∀ a b, own γ (π,a) ⊢ own γ (π/2,a) ∗ own γ (π/2,a).
∀ a b, own γ (π,a) ∗ own γ (π',a) ∗ π + π' <= 1
     ⊢ own γ (π + π',a).
```

If one has `own γ (1,a)`, no other fractions can be owned. Thus one has exclusive ownership and can freely update the resource.

Finite Sets. Given a set X, the finite-set monoid Gset(X) consists of the finite subsets of X under disjoint union. That is, the composition of two finite subsets $a \cdot b$ is defined as the union $a \cup b$ when $a \cap b = \emptyset$, and undefined otherwise.

Finite Maps. Given a set X and monoid Y, the finite-map monoid Gmap(X, Y) consists of the finite partial functions from X to Y. Composition is given by:

$$(a \cdot b)(x) = \begin{cases} a(x) \cdot b(x) & \text{if } x \in \mathrm{dom}(a) \text{ and } x \in \mathrm{dom}(b) \\ a(x) & \text{if } x \in \mathrm{dom}(a) \text{ and } x \notin \mathrm{dom}(b) \\ b(x) & \text{if } x \notin \mathrm{dom}(a) \text{ and } x \in \mathrm{dom}(b) \end{cases}$$

Composition is thus functorial in the co-domain monoid:

```
{[ i := x ]} · {[ i := y ]} = {[ i := x · y ]}
```

where `{[ i := x ]}` represents the singleton map from `i` to `x`.

A frame-preserving update can extend the domain of a finite map with a new key, provided that we are not specific about *which* new key:

$$\checkmark(x) \vdash m \rightsquigarrow \{m' \mid \exists i.\, m' = \{[i := x]\} \cdot m \wedge i \notin \mathrm{dom}(m)\}$$

This gives a way of allocating new ghost resources in a monoid.

File name	Contents
program.v	the Dartino Queue implementation
definitions.v	record definitions that model processes and queues
monoids.v	declarations of ghost resources and lemmas about them
invariants.v	invariants for processes and queues
helpers.v	helper lemmas for the invariants
wp_helpers.v	helper lemmas regarding weakest-precondition reduction of terms
atomize.v	the definition of abstract atomicity
makers.v	proofs for the queue and process constructors
enqueue.v	proofs for enqueuing processes
dequeue.v	proofs for dequeuing processes (at the head and arbitrarily)
client_sequential.v	proofs for a sequential client of the Dartino Queue
client_concurrent.v	proofs for a concurrent client of the Dartino Queue

Table 61: Organization of the Coq project.

Authoritative. Given a monoid X, the authoritative monoid is built from two types of resources: authoritative resources $\bullet a$, and fragment resources $\circ b$. Fragment resources can be composed according to the underlying monoid: $\circ a \cdot \circ b = \circ(a \cdot b)$. Authoritative resources cannot be composed with each other: $\bullet a \cdot \bullet b$ is undefined. When an authoritative and fragment resource are combined, the fragment must be contained with in the authoritative resource: $\checkmark(\bullet a \cdot \circ b)$ implies $b \preccurlyeq a$, where the induced monoid ordering $b \preccurlyeq a$ means that there exists some c such that $b \cdot c = a$.

To perform a frame-preserving update in the authoritative monoid, one typically requires the authoritative resource, and any such update must preserve all fragments that may be owned by other threads. For instance, it is possible to extend the authority by introducing a new fragment:

```
∀ γ a b, own γ •a ∗ ✓(a · b) ⊢ own γ •(a · b) · ○b.
```

The Heap We can now give the ordinary Heap monoid in terms of the above constructions:

$$\textsc{Heap} \triangleq \textsc{Auth}(\textsc{Gmap}(\mathbb{N}, \textsc{Ex}(\mathit{Val})))$$

Having ownership of an authoritative part of a heap is then own $\gamma \bullet$ h, where local ownership of a points-to predicate is written as own $\gamma \circ$ {[l := Excl v]}, which we can give a nicer syntactic representation as l $\mapsto$ v.

6.4 A Specification for the Dartino Queue

In this section, we present the Iris specification for the Dartino Queue. Table 61 shows the structure of the Coq project. While the Coq development includes all proofs, we only present the specifications here.

To simplify our presentation, we take a few liberties with the Coq syntax. In particular, we omit some injections between types (*e.g.* from Coq propositions into Iris terms) as well as scope specifiers.

Datatype Definitions

A reference to a process descriptor object is modelled as a record of four locations, which we refer to as a *process address*. These locations correspond to the addresses of the data members of the object.

```
Record procAddrT := ProcAddrT {
  pvall : loc; pqueuel : loc; pprevl : loc; pnextl : loc
}.
```

A *queue address* is similarly defined as a record of three locations that comprise the data members of a queue object.

```
Record queueAddrT := QueueAddrT {
  qhead : loc; qtail : loc; qsent : loc
}.
```

A *process value* record models the contents of a process descriptor object. It thus comprises the values of each data member of the object.

```
Record procValT := ProcValT' {
  pvalv : val;
  pqueuev : option queueAddrT;
  pprevv : option procAddrT;
  pnextv : option procAddrT
}.
```

Monoids

Our specification of the Dartino Queue uses four custom monoids to represent ghost state, which are explained in detail below.

Process Monoid

The process monoid is the authoritative monoid on partial maps from process addresses to process values with fractional permissions:

$$\textsc{Auth}(\textsc{Gmap}(\texttt{procAddrT}, \textsc{Frac}(\textsc{DecAgree}(\texttt{procValT}))).$$

This monoid represents the current state of process descriptor objects. The authoritative part of the monoid belongs to the invariant `procs_inv`, which ensures that the `pval` pointer of each process matches the value recorded in the monoid. A ¼ fraction of the fragment part typically belongs to an invariant for the process (described by `proc_inv`), which establishes the relationship between the `qref`, `prev` and `next` pointers and the values in the monoid. The remaining ¾ fraction represents ownership of the process, which may either belong to a queue (if the process is in that queue) or a thread.

To denote a fragment part with a given fraction, we define:

```
Definition Proc (x : procAddrT)(π : Qp)(v : procValT) :=
  ○ {[ x := (π, DecAgree v) ]}.
```

Queue Membership Monoid

The queue membership monoid is the authoritative monoid on finite sets of process addresses:

$$\textsc{Auth}(\textsc{Gset}(\texttt{procAddrT})).$$

Each queue has an instance of this monoid that tracks which processes currently belong to it. The authoritative part of the monoid belongs to the predicate that represents a queue, which maintains that the processes recorded in the monoid are exactly those belonging to the queue. The authoritative part is represented as `InQueueAuth l`, where `l` is a list of process addresses. When a process belongs to a queue, the invariant for the process holds a (singleton) fragment of the monoid to track that the process does indeed belong to the queue. This fragment is represented as `InQueue p`, where `p` is a process address. The authoritative monoid gives us the following property:

```
own γ (InQueueAuth l) ∗ own γ (InQueue a) ⊢ a ∈ l
```

Link Monoid

Since queues may be dynamically created, their ghost resources (*i.e.* the queue membership monoid for a queue) are also dynamically allocated. To track these, we use a link monoid that records the ghost resource name associated with queues. This monoid is the authoritative monoid on maps from locations to ghost resource names:

$$\textsc{Auth}(\textsc{Gmap}(\mathit{Loc}, \textsc{DecAgree}(\mathit{Names}))).$$

A fragment part is represented as `Link qs` γ, indicating that the queue with sentinel `qs` is associated with ghost name γ. The authoritative part belongs to a global invariant (`queues`) which tracks the current queues. The following useful property holds for the link monoid:

```
own γq (Link (qsent q) γ) ∗ own γq (Link (qsent q) γ') ⊢ γ = γ'
```

List Monoid

The list monoid is used to track the list of processes that logically belong to a queue. This is used to ensure that, when a thread holds the lock on a queue, no other thread can update the logical contents of the queue: the monoid records the logical contents of the queue; the thread has half of the resource and the queue has the other half; both halves must agree, so only the thread with the lock can update the queue. This monoid is the fractional monoid on lists of process addresses:

$$\textsc{Frac}(\textsc{DecAgree}(\texttt{list procAddrT})).$$

We define `List l` to be a ½ fraction with value `l`. This monoid has the following important property:

$$\texttt{own}\ \gamma\ (\texttt{List}\ l) * \texttt{own}\ \gamma\ (\texttt{List}\ l') \vdash l = l'$$

and, for updating,

$$\texttt{own}\ \gamma\ (\texttt{List}\ l) * \texttt{own}\ \gamma\ (\texttt{List}\ l) \vdash \texttt{own}\ \gamma\ (\texttt{List}\ l') * \texttt{own}\ \gamma\ (\texttt{List}\ l')$$

Predicate Definitions

We now define predicates to represent processes and queues, using the above monoids.

Processes

The `proc` predicate specifies a process:

```
Definition proc γp γq (a : procAddrT) (v : val)
  (qv : option queueAddrT) (pv nv : option procAddrT) :=
  own γp (Proc a ¼ {| pvalv := v; pqueuev := qv;
                      pprevv := pv; pnextv := nv |}) ∗
  (pqueuel a) ↦ option_queueAddrT_to_val qv ∗
  (pprevl a) ↦ option_procAddrT_to_val pv ∗
  (pnextl a) ↦ option_procAddrT_to_val nv ∗
  match qv with
  | None => True
  | Some q => ∃ γ, own γq (Link (qsent q) γ) ∗ own γ (InQueue a)
  end.
```

The assertion `proc γp γq a v qv pv nv` declares ownership of the points-to predicates for the `qref` (`pqueuel a`), `prev` (`pprevl a`) and `next` (`pnextl a`) locations. These locations hold pointers to the specified queue (`qv`), previous process (`pv`) and next process (`nv`) respectively. Furthermore, the assertion declares ownership of a ¼ fragment of the corresponding process ghost resource.

If the process belongs to a queue (*i.e.* `qv` is `Some q`) then the assertion establishes this relationship by holding the ghost resources:

```
own γq (Link (qsent q) γ) * own γ (InQueue a)
```

for some γ. The first of these certifies that the ghost name associated with the queue is γ, while the second certifies that the process logically belongs to the queue.

The predicate `proc_inv γp γq x` wraps the `proc` predicate in an invariant (with other parameters existentially quantified). The `qproc` predicate combines ¾ ownership of the `Proc` ghost resource for a process with the `proc_inv` invariant:

```
Definition qproc γp γq x :=
  ∃ v, own γp (Proc x ¾ {| pvalv := v; pqueuev := None;
                           pprevv := None; pnextv := None
                        |}) ∗ proc_inv γp γq x.
```

Since the `Proc` fragment from the `qproc` predicate must agree with the `Proc` fragment from the `proc_inv` invariant, we can be sure that the heap cells representing the process object will hold the appropriate values. The `qproc` requires that the `qref`, `prev` and `next` pointers should all be `None` — *i.e.* the process does not belong to any queue.

Queues

A queue is represented by the `queue γp γq q γ γ' l` predicate, where `q` is the queue address, `l` is a list of the process addresses for processes that belong to the queue, and the remaining parameters are ghost resource names. A queue may either be locked or unlocked. If it is locked then the queue's `head` pointer must point to the sentinel value, and the majority of the resources representing the queue will have been transferred to the thread that holds the lock. If the queue is unlocked then these resources (which are represented by the `queue_lock` predicate) will belong to the queue. In either state, the queue maintains ½ ownership of the head and sentinel points-to predicates, since threads require access to these in both cases. The predicate also includes one (of two) `List l` ghost resources that tracks the logical contents of the queue; the other is in the `queue_lock` predicate. Finally, it includes a `Link (qsent q) γ'` ghost resource, which records that γ' is the ghost name for the queue's list membership resource.

The predicate is defined as follows:

```
Definition queue γp γq (q : queueAddrT) γ γ' l :=
  ∃ (hv : val) (hvOP tvOP : option procAddrT),
  (qhead q) ↦{1/2} hv ∗ (qsent q) ↦{1/2} () ∗ own γ (List l) ∗
  own γq (Link (qsent q) γ') ∗ (hv = SOMEV (qsent q) ∨
  hv = option_procAddrT_to_val hvOP ∗
  queue_lock γp γq q γ γ' hv hvOP tvOP l).
```

The `queue__lock γp γq q γ γ' hv hv' tv l` predicate represents the majority of the resources that constitute a queue, and which may be obtained by a thread on acquiring the lock. Here, `q` is the queue address, `hv` is the value of the queue's head pointer, `hv'` and `tv` are pointers to the head and tail processes in the queue respectively, and `l` is a list of process addresses that are in the queue.

```
Definition queue_lock γp γq (q : queueAddrT) γ γ' (hv : val)
    (hv' tv : option procAddrT) (l : list procAddrT) :=
  (qhead q) ↦{1/2} hv ∗ own γ (List l) ∗ own γ' (InQueueAuth l) ∗
  (qtail q) ↦ option_procAddrT_to_val tv ∗ queue_cont γp γq q hv' tv l.
```

The `queue__lock` predicate includes half ownership of the queue's head pointer and the second `List l` ghost resource for the queue. These resources complement those of the `queue` predicate, and ensure that the head pointer and logical contents of the queue cannot be changed by other threads while the lock is held.

The predicate also asserts ownership of `InQueueAuth l` ghost resource, which ensures that only processes in `l` can have a corresponding `InQueue` resource. Moreover, the full permission on the queue's tail pointer belongs to the `queue__lock` predicate, since this pointer is only accessed by threads that have acquired the lock. Finally, the list of processes, represented by the `queue__cont` predicate, completes the predicate.

The predicate `queue__cont` consists of `proc__inv` invariants for each process in the list, together with a recursively-defined predicate `queue__dll` that ensures that the processes form a doubly-linked list.

```
Definition queue_cont γp γq (q : queueAddrT)
    (h t : option procAddrT) (l : list procAddrT) :=
  ([∗ list] p ∈ l, proc_inv γp γq p) ∗ queue_dll γp q l h t None None.
```

The `queue__dll γp q l i e p n` resembles a standard doubly-linked list segment predicate [66], except that `Proc` ghost resources are used to represent the nodes of the list. (The `proc__inv` invariant for each process establishes the connection between these ghost resources and the actual values in the process object, since it holds the complementary `Proc` resource.) The pointers `i` and `e` are to the first and last processes in the segment, respectively, and `p` and `n` are the previous and next pointers of the first and last nodes of the segment.

```
Fixpoint queue_dll γp (q : queueAddrT) (l : list procAddrT)
    (i e p n : option procAddrT) :=
  match l with
  | nil => (i = n ∧ e = p)
  | x :: l' => ∃ (v : val) (n' : option procAddrT),
               i = Some x ∗ own γp (Proc x ¾ {| pvalv := v;
                                               pqueuev := (Some q);
                                               pprevv := p;
                                               pnextv := n' |}) ∗
               queue_dll γp q l' n' e i n
  end.
```

6.5 A Logically Atomic Specification for the Dartino Queue

One approach to specifying the Dartino Queue would be with Hoare-triples such as the following[1]:

$$\{\mathsf{qproc}\ p * \mathsf{queue}\ q\ l\}$$
$$\quad \mathsf{enqueue}(q, p)$$
$$\{v.\ v = () * \mathsf{queue}\ q\ (l +\!\!+ [p])\}$$

This specifies that calling enqueue with a valid queue q and un-enqueued process p will result in the process being appended to the queue. Unfortunately, to use this specification, a thread must have ownership of the queue. Therefore, it is not useful in a concurrent situation where the queue may be shared among many threads (such as a scheduler).

An alternative specification would be to wrap the queue in an invariant:

$$\{\mathsf{qproc}\ p * \mathsf{inv}\ N\ (\exists l.\ \mathsf{queue}\ q\ l)\}$$
$$\quad \mathsf{enqueue}(q, p)$$
$$\{v.\ v = () * \mathsf{inv}\ N\ (\exists l.\ \mathsf{queue}\ q\ l)\}$$

With such a specification, multiple threads can access the queue. However, we lose the information that enqueue actually appends the process to the queue. Indeed an implementation could not change the queue at all and be correct with respect to such a specification.

The problem with the first specification is that we do not allow any concurrent updates to the queue. The problem with the second is that we allow all possible concurrent updates to the queue. The optimal specification would allow the client of the queue to determine exactly which concurrent updates

[1] For exposition, we elide some parameters of the predicates.

are possible. We can achieve such a specification by viewing the update as *logically atomic* [18].

Access to invariants is generally only permitted to atomic operations: if the operation preserves the invariant, then no other thread can observe a violation of the invariant because the operation is atomic. Logically atomic operations can similarly be used to access invariants, although they do not execute in a single atomic step. In [18], da Rocha Pinto *et al.* propose an *atomic triple* for specifying logical atomicity. For `enqueue`, we might give the following atomic triple:

$$\begin{array}{c} \mathbb{V}l.\langle \mathsf{qproc}\ p * \mathsf{queue}\ q\ l \rangle \\ \mathsf{enqueue}(q,p) \\ \langle v.\ v = () * \mathsf{queue}\ q\ (l +\!\!+ [p]) \rangle \end{array}$$

This specification expresses that the process `p` is atomically appended to the queue `q` in the execution of `enqueue(q,p)`. The binding of `l`, representing the contents of the queue, allows the client to arbitrarily update the queue during the execution of `enqueue`, provided that the precondition holds for *some* `l` up until the atomic update takes effect. Immediately after the atomic update, the postcondition will hold for the value of `l` at which the precondition held immediately prior.

Logical Atomicity in Iris

In Iris hoare-triples are encoded using weakest precondition, as so:

$$\{P\}\, e\, \{\Phi\} \triangleq \Box(P \mathrel{-\!\!*} \mathsf{wp}\, e \left\{\Phi\right\})$$

Therefore, we show how to construct a logically-atomic weakest precondition in Iris. The core idea is expressed by an "atomic shift" [33]:

```
Definition atomic_shift {A B : Type}
  (α: A → iProp Σ)      (* atomic pre-condition *)
  (β: A → B → iProp Σ) (* atomic post-condition *)
  (Ei Eo: coPset)       (* inside/outside invs *)
  (P : iProp Σ) (Q : A → B → iProp Σ) : iProp Σ :=
    (P ={Eo, Ei}=> ∃ x:A, α x ∗
      ((α x ={Ei, Eo}=∗ P) ∧ (∀ y, β x y ={Ei, Eo}=∗ Q x y))).
```

An atomic shift is a persistent assertion. It effectively captures that an atomic update from α to β is sufficient to take precondition `P` to postcondition `Q`. Specifically, it says:

- From the precondition `P` we can obtain α `x` for some `x`, by opening the invariants `Eo` \ `Ei`.
- Having done so, it is possible to restore `P` by reestablishing α `x` and closing the invariants.

- Alternatively, by instead establishing β x y for any y, we my establish Q x y by closing the invariants.

The idea is that if an operation e performs a logically-atomic update from α to β, then for any given P and Q such that atomic_shift α β Ei Eo P Q we have {P} e {Q}. Such an operation thus consists of steps that may access α x but must preserve it, followed by a step that updates α x to β x y, followed by steps that cannot violate the (arbitrary) postcondition Q. This idea is expressed in the definition of logically-atomic weakest precondition:

```
Definition atomic_wp {A : Type}
  (α: A → iProp Σ)          (* atomic pre-cond. *)
  (β: A → val _ → iProp Σ) (* atomic post-cond. *)
  (Ei Eo : coPset)          (* in/out invs *)
  (e: expr _) : iProp Σ :=
  (∀ P Q, atomic_shift α β Ei Eo P Q -∗
    P -∗ WP e {{ v, ∃ x: A, Q x v }}).
```

Logically-atomic Specifications for the Dartino Queue Operations

Using this definition of logical atomicity, we can finally show the following specification for enqueue:

```
Lemma enqueue_spec :
  ∀ q p γ γ' E, procs_inv N γp ∗ qproc N γp γq p ⊢
  atomic_wp (λ l => ▷ queue N γp γq q γ γ' l)
            (λ l ret => queue N γp γq q γ γ' (l++[p]) ∗
                        ret = is_nil l)
            ∅ E
            enqueue (qhead q) (qtail q) (qsent q) (procAddrT_to_val p).
```

This specification establishes that enqueue atomically adds the process p to the end of the queue q, with the return value indicating whether the queue was empty at the time. However, the qproc predicate does not form part of the atomic precondition. Instead, it is in the overall precondition. This means that ownership of the qproc predicate is transferred to enqueue when it is called, rather than at the point it performs the atomic update. (This is analogous to the generalization of atomic triples in [18] that permits this kind of resource transfer.) The implementation can thus use these particular resources as it sees fit, without being concerned with interference from other threads. The overall precondition also establishes the invariant procs_inv.

Note that the atomic precondition is guarded under the ▷ modality. Since we have $P \vdash \triangleright P$, we could derive a specification without ▷. However, in Iris when an invariant is opened with a view shift, the contents is guarded by the ▷ modality. Therefore it is more convenient for clients that the atomic precondition should be guarded by ▷.

We can also show the following specification for dequeue:

```
Lemma dequeue_spec :
  ∀ q γ γ' E, procs_inv N γp ⊢
  atomic_wp (λ l => ▷ queue N γp γq q γ γ' l)
            (λ l ret => ∃ p l', l = p :: l' ∗ queue N γp γq q γ γ' l' ∗
                        qproc N γp γq p ∗ ret = Some p ∨
                        l = [] ∗ queue N γp γq q γ γ' [] ∗ ret = NONE)
            ∅ E
            dequeue (qhead q) (qtail q) (qsent q) ().
```

Note that the atomic postcondition consists of a disjunction of the two cases: either the queue was non-empty and the process at the head of the queue is dequeued and returned; or the queue was empty, it is unchanged and the value `NONE` is returned.

Finally, we present the specification for `tryDequeueEntry`:

```
Lemma tryDequeueEntry_spec :
  ∀ q p γ γ' E, procs_inv N γp ∗ proc_inv N γp γq p ⊢
  atomic_wp (λ l => ▷ queue N γp γq q γ γ' l)
            (λ l ret => (p ∈ l ∗ ∃ l1 l2, l = l1 ++ p :: l2 ∗
                        queue N γp γq q γ γ' (l1 ++ l2) ∗
                        qproc N γp γq p ∗ ret = true) ∨
                        (p ∉ l ∗ queue N γp γq q γ γ' l ∗ ret = false))
            (n_inv_proc N p) E
            tryDequeueEntry (qhead q) (qtail q) (qsent q)
                            (procAddrT_to_val p).
```

As with `dequeue`, the atomic postcondition is a disjunction: if the process `p` is in `l`, then the list `l` can be split such that $l = l1 ++ p :: l2$, so we update the queue to $l1 ++ l2$, extract the `qproc` resource for `p`, and return `true`; if `p` is not in `l`, we do nothing and return `false`.

Note that the global precondition requires the `proc_inv` invariant for the process we wish to dequeue, in addition to the `procs_inv` invariant present in the other specifications. This is since otherwise we would have no guarantee that `p` indeed represents a legitimate process object.

Interestingly, we also require that the invariant for the process is closed when obtaining the atomic pre-condition. This is because we have to case on the process being in the queue, which requires us to open the invariant. Since it is unsound to open the invariant twice, we need to enforce that the client does not open the invariant for the process.

6.6 Client

Logically atomic specifications allow clients to build and enforce their own protocol on top of data-structures. To illustrate this, we will consider a simple client of the Dartino Queue that simulates a round-robin scheduler. To simulate executing a process, we define a function `doWork` that simply reads

and writes a process's pval pointer. We also define a function scheduler, which loops attempting to dequeue, "execute" and re-enqueue a process from a given queue, and a function enqueuer, which loops creating fresh processes and enqueuing them.

```
Definition doWork : val := λ: pval, pval <- !pval.

Definition scheduler : val :=
  rec: loop h t s :=
    let: p := dequeue h t s () in
    match: p with
      NONE => ()
    | SOME p' => doWork (pval p');;
                 enqueue h t s p'
    end;;
    loop h t s.

Definition enqueuer : val :=
  rec: loop h t s :=
    let: p := makeProc 1 in
    enqueue h t s p;;
    loop h t s.

Definition concurrent_client : val :=
  λ: <>,
     let: q1 := makeQueue () in
     let: q1h := queue_head q1 in
     let: q1t := queue_tail q1 in
     let: q1s := queue_sent q1 in
     Fork (scheduler q1h q1t q1s);;
     Fork (scheduler q1h q1t q1s);;
     Fork (enqueuer q1h q1t q1s).
```

The concurrent_client function creates a new Dartino Queue and forks two scheduler threads to run processes from the queue, and one enqueuer thread to add processes to the queue. (Recall that queue_head, queue_tail and queue_sent are projection functions from the tuple that represents a queue address.)

For doWork, to read and write a process's pval member, our custom protocol needs to transfer ownership of the reference to doWork. Similarly, for enqueuer and scheduler to operate on the same queue, the protocol should allow for each to access the queue, to transfer ownership of the process's pval to the shared state when enqueueing a process, and to remove the ownership of pval when dequeuing a process.

The following invariant client_queue_inv is an excellent candidate for the shared state for our custom protocol:

```
Definition client_queue γp γq q γ γ' l :=
  queue N γp γq q γ γ' l ∗ [∗ list] p ∈ l, ∃ v, pvall p ↦{1/2} v
```

```
Definition client_queue_inv γp γq (q : queueAddrT) γ γ' :=
  inv (n_inv_queue q) (∃ l, client_queue γp γq q γ γ' l).
```

This invariant holds a ½ fraction of the `pval` pointer for each process in the queue. (The remaining ½ belongs to the `procs_inv` invariant, and can be obtained from the `qproc` resource when a process is removed from the queue.)

To show how this custom protocol works, consider how the `scheduler` function will use the logically atomic specification for `dequeue`. To do so, it must establish an `atomic_shift α β Ei Eo P Q`, where α, β and `Ei` are determined by the `dequeue` specification as:

```
α := (λ l => ▷ queue N γp γq q γ γ' l)
β := (λ l ret => ∃ p l', l = p :: l' ∗ queue N γp γq q γ γ' l' ∗
                 qproc N γp γq p ∗ ret = Some p ∨
                 l = [] ∗ queue N γp γq q γ γ' [] ∗ ret = NONE)
Ei := ∅
```

and `Eo`, `P` and `Q` are determined by the client as:

```
Eo := { n_inv_queue q }
P := True
Q := λ l ret => l = [] ∗ ret = NONE ∨
                ∃ p l' v, l = p :: l' ∗ qproc N γp γq p ∗
                          ▷(pval p) ↦{1/2} v ∗ ret = Some p
```

The precondition is `True` since the queue belongs to the client invariant, which is persistent. The postcondition extracts the `qproc` and `pval` pointer resources from the queue when the operation succeeds. The client obtains `P` and `Q` as a pre- and postcondition for `dequeue` by establishing the atomic shift, namely:

```
P ={Eo, Ei}=> ∃ x:A, α x ∗
  (α x ={Ei, Eo}=∗ P) ∧ (∀ y, β x y ={Ei, Eo}=∗ Q x y))
```

Since `Eo` is `{ n_inv_queue q }` and `Ei` is $\emptyset$, the view shift opens the client invariant to obtain α. Recall that opening an invariant obtains its resources guarded under the later modality ($\triangleright$), and hence its presence in α. The wand shifts close the client invariant, the first when no update is performed, and the second when the `dequeue` operation takes effect. For the latter, when the operation succeeds the dequeued process is no longer in the queue, and we have

```
queue N γp γq q γ γ' l ∗ ▷ [∗ list] p' ∈ p::l, ∃ v, pvall p' ↦{1/2} v
```

To close the invariant again, we unfold the iterated separating conjunction over lists (`[* list]`) to extract the `pval` resource for the process `p` that is being dequeued:

```
queue N γp γq q γ γ' l ∗ ∃ v', ▷ pvall p' ↦{1/2} v' ∗
▷ [∗ list] p' ∈ l, ∃ v, pvall p' ↦{1/2} v
```

The client invariant can then be closed and the ▷ `pvall p' ↦{½} v'` resource can be retained by the postcondition `Q`.

6.7 Conclusion

We have formally specified and verified the concurrent queue data structure at the heart of the Dartino Framework using Iris in the Coq proof assistant. While the algorithm itself is fairly simple, giving a reasonable specification for it is not trivial. For this, we have used an encoding of logical atomicity in Iris. Logical atomicity allows us to precisely capture the behaviour of the queue operations, allowing clients of the data structure to impose their own invariants. We demonstrate this by verifying a concurrent client using our specification. Our work is a case study which shows that Iris and logical atomicity can be effectively applied to reason about real-world code.

Appendix

A Relational Model of Types-and-Effects in Higher-Order Concurrent Separation Logic

The Language and Typing Rules

Syntax and Operational Semantics of $\lambda_{ref,conc}$

The syntax of $\lambda_{ref,conc}$ is shown in Figure 1 and the operational semantics is presented in Figure 2. We assume given denumerably infinite sets of variables Var, ranged over by x, y, f, and locations Loc, ranged over by l. We use v to range over the set of values, Val, and e to range over the set of expressions, Exp. Note that expressions do not include types.

Heaps are finite partial maps from Loc to Val and a thread-pool is a finite partial map from thread identifiers, modeled by natural numbers $\mathbb{N}$, to expressions Exp.

The operational semantics is defined by a small-step relation between configurations consisting of a heap and a thread-pool, where each individual step of the system is either a reduction on a thread or the forking of a new thread. The semantics is defined in terms of evaluation contexts, $K \in \mathrm{ECtx}$. We use $K[e]$ to denote the expression obtained by plugging e into the context

$$
\begin{aligned}
\mathrm{Val}\ v &::= () \mid n \mid (v, v) \mid \mathsf{inj}_i\ v \mid \mathsf{rec}\ f(x).e \mid x \mid l \\
\mathrm{Exp}\ e &::= v \mid e = e \mid e\ e \mid (e, e) \mid \mathsf{prj}_i\ e \mid \mathsf{inj}_i\ e \mid e + e \\
&\quad \mid \mathsf{case}(e, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \\
&\quad \mid \mathsf{new}\ e \mid !e \mid e := e \mid \mathsf{CAS}(e, e, e) \mid e \| e
\end{aligned}
$$

Figure 1: Syntax of $\lambda_{ref,conc}$.

$$
\begin{array}{l}
\textsc{Heap}\ h \in \textsc{Loc} \xrightarrow{\text{fin}} \textsc{Val} \\
\textsc{ECtx}\ K ::= [\,] \mid K = e \mid v = K \mid K\ e \mid v\ K \mid (K, e) \mid (v, K) \\
\quad \mid\ \mathsf{prj}_i\ K \mid \mathsf{inj}_i\ K \mid K + e \mid v + K \mid \mathsf{case}(K, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \\
\quad \mid\ \mathsf{new}\ K \mid\ !K \mid K := e \mid v := K \mid K \| e \mid e \| K \\
\quad \mid\ \mathsf{CAS}(K, e, e) \mid \mathsf{CAS}(v, K, e) \mid \mathsf{CAS}(v, v, K)
\end{array}
$$

Pure reduction $\boxed{e \xrightarrow{\text{pure}} e'}$

$$
\begin{array}{c}
(\mathsf{rec}\ f(x).e)\ v \xrightarrow{\text{pure}} e[v/x, \mathsf{rec}\ f(x).e/f] \\
\mathsf{case}(\mathsf{inj}_i\ v, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ x \Rightarrow e_2) \xrightarrow{\text{pure}} e_i[v/x] \\
v_1 \| v_2 \xrightarrow{\text{pure}} (v_1, v_2) \qquad \mathsf{prj}_i\ (v_1, v_2) \xrightarrow{\text{pure}} v_i \\
v_1 + v_2 \xrightarrow{\text{pure}} v_3 \quad \text{where } v_3 = v_1 + v_2 \\
v = v \xrightarrow{\text{pure}} \mathsf{true} \qquad v_1 = v_2 \xrightarrow{\text{pure}} \mathsf{false} \qquad \text{where } v_1 \neq v_2
\end{array}
$$

Reduction $\boxed{h; e \to h'; e'}$

$$
\begin{array}{rll}
h; e \to h; e' & & \text{if } e \xrightarrow{\text{pure}} e' \\
h; \mathsf{new}\ v \to h \uplus [l \mapsto v]; l & & \\
h; !l \to h; v & & \text{if } h(l) = v \\
h[l \mapsto -]; l := v \to h[l \mapsto v]; () & & \\
h; \mathsf{CAS}(l, v_o, v_n) \to h; \mathsf{false} & & \text{if } h(l) \neq v_o \\
h[l \mapsto v_o]; \mathsf{CAS}(l, v_o, v_n) \to h[l \mapsto v_n]; \mathsf{true} & & \\
h; K[e] \to h'; K[e'] & & \text{if } h; e \to h'; e'
\end{array}
$$

Figure 2: Operational semantics of $\lambda_{ref,conc}$.

K and $e[v/x]$ to denote capture-avoiding substitution of value v for variable x in expression e.

Typing rules

We assume a denumerably infinite set RegVar of region variables, ranged over by ρ. An atomic effect on a region ρ is either a read effect, rd_ρ, a write effect, wr_ρ, or an allocation effect, al_ρ. An effect ε is a finite set of atomic effects. The

set of types is defined by the following grammar:

$$\textsc{Type}\ \tau ::= \mathbf{1} \mid \mathsf{int} \mid \mathsf{ref}_\rho\ \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau \to^{\Pi,\Lambda}_{\varepsilon} \tau$$

where Π and Λ are finite sequences of region variables. Typing judgments take the form

$$\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$$

Monoids and Constructions

Evaluation Context Monoid

Extended Expressions

$\boxed{\mathcal{E} \in EExp}$

$$\begin{aligned} \mathcal{E} \in EExp ::= {} & a \mid () \mid n \mid x \mid l \mid \mathsf{rec}\ f(x).e \mid \mathcal{E} = \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid (\mathcal{E}, \mathcal{E}) \mid \mathcal{E} + \mathcal{E} \\ \mid {} & \mathsf{prj}_i\ \mathcal{E} \mid \mathsf{inj}_i\ \mathcal{E} \mid \mathsf{case}(\mathcal{E}, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \\ \mid {} & \mathsf{new}\ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := \mathcal{E} \mid \mathsf{CAS}(\mathcal{E}, \mathcal{E}, \mathcal{E}) \mid \mathcal{E} \| \mathcal{E} \end{aligned}$$

where $a \in \mathcal{A}$ is an address.

Extended Evaluation Contexts

$\boxed{\kappa \in EECtx}$

$$\begin{aligned} \kappa \in EECtx ::= {} & \bullet \mid \kappa = \mathcal{E} \mid v = \kappa \mid \kappa\ \mathcal{E} \mid v\ \kappa \mid (\kappa, \mathcal{E}) \mid (v, \kappa) \mid \kappa + \mathcal{E} \mid v + \kappa \\ \mid {} & \mathsf{prj}_i\ \kappa \mid \mathsf{inj}_i\ \kappa \mid \mathsf{case}(\kappa, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \\ \mid {} & \mathsf{new}\ \kappa \mid !\kappa \mid \kappa := \mathcal{E} \mid v := \kappa \\ \mid {} & \kappa \| \mathcal{E} \mid \mathcal{E} \| \kappa \mid \mathsf{CAS}(\kappa, \mathcal{E}, \mathcal{E}) \mid \mathsf{CAS}(v, \kappa, \mathcal{E}) \mid \mathsf{CAS}(v, v, \kappa) \end{aligned}$$

Multi Evaluation Contexts

$\boxed{MECtx \subseteq EExp}$

$$\begin{aligned} B \in MECtx ::= {} & a \mid e \mid B = e \mid v = B \mid B\ e \mid v\ B \mid (B, e) \mid (v, B) \mid B + e \\ \mid {} & v + B \mid \mathsf{prj}_i\ B \mid \mathsf{inj}_i\ B \mid \mathsf{case}(B, \mathsf{inj}_1\ x \Rightarrow e, \mathsf{inj}_2\ y \Rightarrow e) \\ \mid {} & \mathsf{new}\ B \mid !B \mid B := e \mid v := B \\ \mid {} & B \| B \mid \mathsf{CAS}(B, e, e) \mid \mathsf{CAS}(v, B, e) \mid \mathsf{CAS}(v, v, B) \end{aligned}$$

Free Addresses

$\boxed{FA : EExp \rightharpoonup \mathcal{P}(\mathcal{A})}$

$$\frac{}{\Pi \mid \Lambda \mid \Gamma, x : \tau \vdash x : \tau, \emptyset} \qquad \frac{}{\Pi \mid \Lambda \mid \Gamma \vdash () : \mathbf{1}, \emptyset} \qquad \frac{v \in \{\mathbf{true}, \mathbf{false}\}}{\Pi \mid \Lambda \mid \Gamma \vdash v : \mathbf{B}, \emptyset}$$

$$\frac{v \in \mathbb{N}}{\Pi \mid \Lambda \mid \Gamma \vdash v : \mathbf{int}, \emptyset} \qquad \frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_i, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{inj}_i\ e : \tau_1 + \tau_2, \varepsilon}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \tau, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \tau, \varepsilon_2 \qquad eq_{type}(\tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e_1 = e_2 : \mathbf{B}, \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_1 + \tau_2, \varepsilon \qquad \Pi \mid \Lambda \mid \Gamma, x_i : \tau_i \vdash e_i : \tau, \varepsilon_i}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{case}(e, \mathbf{inj}_1\ x_1 \Rightarrow e_1, \mathbf{inj}_2\ x_2 \Rightarrow e_2) : \mathbf{B}, \varepsilon \cup \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_1 \times \tau_2, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{prj}_i\ e : \tau_i, \varepsilon} \qquad \frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \mathbf{int}, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \mathbf{int}, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e_1 + e_2 : \mathbf{int}, \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma, f : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, x : \tau_1 \vdash e : \tau_2, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{rec}\ f(x).e : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, \emptyset}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \tau_1, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e_1\ e_2 : \tau_2, \varepsilon \cup \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon \qquad \rho \in \Pi, \Lambda}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{new}\ e : \mathbf{ref}_{\rho}\ \tau, \varepsilon \cup \{al_{\rho}\}}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \mathbf{ref}_{\rho}\ \tau, \varepsilon_1 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \tau, \varepsilon_2}{\Pi \mid \Lambda \mid \Gamma \vdash e_1 := e_2 : \mathbf{1}, \varepsilon_1 \cup \varepsilon_2 \cup \{wr_{\rho}\}} \qquad \frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \mathbf{ref}_{\rho}\ \tau, \varepsilon}{\Pi \mid \Lambda \mid \Gamma \vdash !e : \tau, \varepsilon \cup \{rd_{\rho}\}}$$

$$\frac{\Pi \mid \Lambda, \rho \mid \Gamma \vdash e : \tau, \varepsilon \qquad \rho \notin FRV(\Gamma, \tau)}{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon - \rho}$$

$$\frac{\Pi, \Lambda_3 \mid \Lambda_1 \mid \Gamma_1 \vdash e_1 : \tau_1, \varepsilon_1 \qquad \Pi, \Lambda_3 \mid \Lambda_2 \mid \Gamma_2 \vdash e_2 : \tau_2, \varepsilon_2}{\Pi \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma_1, \Gamma_2 \vdash e_1 \| e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\begin{array}{c}\Pi \mid \Lambda \mid \Gamma \vdash e_1 : \mathbf{ref}_{\rho}\ \tau, \varepsilon_1 \\ \Pi \mid \Lambda \mid \Gamma \vdash e_2 : \tau, \varepsilon_2 \qquad \Pi \mid \Lambda \mid \Gamma \vdash e_3 : \tau, \varepsilon_3 \qquad eq_{type}(\tau)\end{array}}{\Pi \mid \Lambda \mid \Gamma \vdash \mathbf{CAS}(e_1, e_2, e_3) : \mathbf{B}, \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3 \cup \{wr_{\rho}, rd_{\rho}\}}$$

$$\frac{}{eq_{type}(\mathbf{1})}$$

$$\frac{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_1, \varepsilon_1 \qquad \Pi, \Lambda \vdash \tau_1 \leq \tau_2 \qquad \varepsilon_1 \subseteq \varepsilon_2 \qquad FRV(\varepsilon_2) \in \Pi, \Lambda}{\Pi \mid \Lambda \mid \Gamma \vdash e : \tau_2, \varepsilon_2}$$

$$\frac{eq_{type}(\tau) \qquad eq_{type}(\sigma) \qquad op \in \{+, \times\}}{eq_{type}(\tau \; op \; \sigma)} \qquad \frac{FRV(\tau) \in \Pi \cup \Lambda}{\Pi \cup \Lambda \vdash \tau \leq \tau}$$

$$\frac{\Pi \cup \Lambda \vdash \tau_1 \leq \tau_1' \qquad \Pi \cup \Lambda \vdash \tau_2 \leq \tau_2'}{\Pi \cup \Lambda \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$\frac{\Pi \cup \Lambda \vdash \tau_1 \leq \tau_1' \qquad \Pi \cup \Lambda \vdash \tau_2 \leq \tau_2' \qquad \varepsilon_1 \subseteq \varepsilon_2 \qquad \Pi_1 \subseteq \Pi_2 \qquad \Lambda_1 \subseteq \Lambda_2}{\Pi \cup \Lambda \vdash \tau_1 \to^{\Pi_1, \Lambda_1}_{\varepsilon_1} \tau_2 \leq \tau_1' \to^{\Pi_2, \Lambda_2}_{\varepsilon_2} \tau_2'}$$

Figure 2: Typing and sub-typing inference rules. We write $FV(e)$ and $FRV(e)$ for the sets of free program variables and region variables respectively. For all typing judgments on the form $\Pi \mid \Lambda \mid \Gamma \vdash e : \tau, \varepsilon$ we always have $FRV(\Gamma, \tau, \varepsilon) \in \Pi \cup \Lambda$. The equality type predicate, eq_{type}, defines the types we may test for equality.

$$\begin{aligned}
FA(a) &\triangleq \{a\} \\
FA(()) = FA(x) = FA(l) = FA(\mathbf{rec}\ f(x).e) &\triangleq \emptyset \\
FA(\mathbf{prj}_i\ \mathcal{E}) = FA(\mathbf{inj}_i\ \mathcal{E}) = FA(\mathbf{new}\ \mathcal{E}) = FA(!\mathcal{E}) &\triangleq FA(\mathcal{E}) \\
FA(\mathbf{case}(\kappa, \mathbf{inj}_1\ x \Rightarrow e_1, \mathbf{inj}_2\ y \Rightarrow e_2)) &\triangleq FA(\mathcal{E}) \\
FA(\mathcal{E}_1 = \mathcal{E}_2) = FA(\mathcal{E}_1\ \mathcal{E}_2) &\triangleq FA(\mathcal{E}_1) \uplus FA(\mathcal{E}_2) \\
FA(\mathcal{E}_1 := \mathcal{E}_2) = FA(\mathcal{E}_1 \parallel \mathcal{E}_2) &\triangleq FA(\mathcal{E}_1) \uplus FA(\mathcal{E}_2) \\
FA((\mathcal{E}_1, \mathcal{E}_2)) = FA(\mathcal{E}_1 + \mathcal{E}_2) &\triangleq FA(\mathcal{E}_1) \uplus FA(\mathcal{E}_2) \\
FA(\mathbf{CAS}(\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3)) &\triangleq FA(\mathcal{E}_1) \uplus FA(\mathcal{E}_2) \uplus FA(\mathcal{E}_3)
\end{aligned}$$

where $A \uplus B$ is the union of A and B, but is only defined if A and B are disjoint.

Evaluation Context Monoid ECtx

$$\text{ECtx} \triangleq (\{f : \mathcal{A} \rightharpoonup_{fin} MECtx \mid \forall a \in \text{dom}(f).\, \forall b \in FA(f(a)).\, a <_{\mathcal{A}} b\}, \cdot, [])$$

where $<_{\mathcal{A}}$ is strict ordering on addresses and monoid composition is defined as follows

$$f \cdot g \triangleq \begin{cases} \bot & \text{if } \mathrm{dom}(f) \cap \mathrm{dom}(g) \neq \emptyset \\ f \cup g & \text{otherwise} \end{cases}$$

Hereditarily Free Addresses $\boxed{FA : EExp \times |\text{ECTX}| \rightharpoonup \mathcal{P}(\mathcal{A})}$

$$FA(\mathcal{E}, f) \triangleq FA(\mathcal{E}) \uplus \biguplus \{FA(f(a), f \setminus \{a\}) \mid a \in FA(\mathcal{E}) \cap \mathrm{dom}(f)\}$$

The $FA(\mathcal{E}, f)$ function is defined by recursively on the size of (the domain of) f.

Address Substitution $\boxed{subst : EExp \times |\text{ECTX}| \to EExp}$

$$\begin{aligned}
subst(a)(f) &\triangleq \begin{cases} subst(f(a), f \setminus \{a\}) & \text{if } a \in \mathrm{dom}(f) \\ a & \text{otherwise} \end{cases} \\
subst(e, f) &\triangleq e \\
subst(\mathcal{E}_1 = \mathcal{E}_2, f) &\triangleq subst(\mathcal{E}_1, f) = subst(\mathcal{E}_2, f) \\
subst(\mathcal{E}_1\ \mathcal{E}_2, f) &\triangleq subst(\mathcal{E}_1, f)\ subst(\mathcal{E}_2, f) \\
subst((\mathcal{E}_1, \mathcal{E}_2), f) &\triangleq (subst(\mathcal{E}_1, f), subst(\mathcal{E}_2, f)) \\
subst(\mathcal{E}_1 + \mathcal{E}_2, f) &\triangleq subst(\mathcal{E}_1, f) + subst(\mathcal{E}_2, f) \\
subst(\mathsf{prj}_i\ \mathcal{E}, f) &\triangleq \mathsf{prj}_i\ subst(\mathcal{E}, f) \\
subst(\mathsf{inj}_i\ \mathcal{E}, f) &\triangleq \mathsf{inj}_i\ subst(\mathcal{E}, f) \\
subst(\mathsf{case}(\kappa, \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ y \Rightarrow e_2), f) & \\
&\triangleq \mathsf{case}(subst(\kappa, f), \mathsf{inj}_1\ x \Rightarrow e_1, \mathsf{inj}_2\ y \Rightarrow e_2) \\
subst(\mathsf{new}\ \mathcal{E}, f) &\triangleq \mathsf{new}\ subst(\mathcal{E}, f) \\
subst(!\mathcal{E}, f) &\triangleq !subst(\mathcal{E}, f) \\
subst(\mathcal{E}_1 := \mathcal{E}_2, f) &\triangleq subst(\mathcal{E}_1, f) := subst(\mathcal{E}_2, f) \\
subst(\mathsf{CAS}(\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3), f) &\triangleq \mathsf{CAS}(subst(\mathcal{E}_1, f), subst(\mathcal{E}_2, f), subst(\mathcal{E}_3, f)) \\
subst(\mathcal{E}_1 \| \mathcal{E}_2, f) &\triangleq subst(\mathcal{E}_1, f) \| subst(\mathcal{E}_2, f)
\end{aligned}$$

The $subst(\mathcal{E}, f)$ function is defined by lexicographic recursion on the size of f and $\mathcal{E}$.

Extended Context Substitution $\boxed{-[=] : EECtx \times Exp \to MECtx}$

The extended context substitution function, $\kappa[e]$, substitues the expression e for the $\bullet$ in κ in the obvious way.

Lemma .0.1.

$$\forall \mathcal{E}.\, \forall f \in |\text{ECtx}|.\, \forall a \in FA(subst(\mathcal{E}, f)).\, \exists b \in FA(\mathcal{E}).\, b \leq_{\mathcal{A}} a$$

Proof. By lexicographic induction on $|f|$ and the size of $\mathcal{E}$.

- Case $\mathcal{E} = c$: if $c \in \text{dom}(f)$ then $subst(\mathcal{E}, f) = subst(f(c), f \setminus \{c\})$ and it follows by the induction hypothesis that there exists a $b \in FA(f(c))$ such that $b \leq_{\mathcal{A}} a$. Furthermore, by definition of $|\text{ECtx}|$ it follows that $c < b$ and thus by transitivity that $c <_{\mathcal{A}} a$ and $c \in FA(\mathcal{E})$.

  Conversely, if $c \notin \text{dom}(f)$ then $subst(\mathcal{E}, f) = \mathcal{E}$ and it follows trivially by choosing $b = a$.

- All remaining cases follow directly from the induction hypothesis.

□

Lemma .0.2.

$$\forall \mathcal{E}.\, subst(\mathcal{E}, []) = \mathcal{E}$$

Lemma .0.3.

$$\begin{aligned}&\forall \mathcal{E}.\, \forall f_1, f_2 \in |\text{ECtx}|.\\ &\quad (\forall a \in FA(\mathcal{E}).\, \forall b \geq_{\mathcal{A}} a.\, (b \in \text{dom}(f_1) \Leftrightarrow\\ &\qquad b \in \text{dom}(f_2)) \wedge f_1(b) = f_2(b)) \Rightarrow subst(\mathcal{E}, f_1) = subst(\mathcal{E}, f_2)\end{aligned}$$

Proof. By lexicographic induction on $|f_1|$ and the size of $\mathcal{E}$.

- Case $\mathcal{E} \cong a$: then $a \in FA(\mathcal{E})$. If $a \in \text{dom}(f_1)$ then $a \in \text{dom}(f_2)$, $f_1(a) = f_2(a)$ and thus,

  $$\begin{aligned}subst(\mathcal{E}, f_1) = subst(f_1(a), f_1) \overset{IH}{=} subst(f_1(a), f_2) =\\ subst(f_2(a), f_2) = subst(\mathcal{E}, f_2)\end{aligned}$$

  and if $a \notin \text{dom}(f_1)$, then $a \notin \text{dom}(f_2)$ and thus

  $$subst(\mathcal{E}, f_1) = a = subst(\mathcal{E}, f_2)$$

- All the remaining cases follow directly from the induction hypothesis.

□

Definition .0.4.

$$f =_a g \triangleq \forall b >_{\mathcal{A}} a.\, (b \in \mathrm{dom}(f) \Leftrightarrow b \in \mathrm{dom}(g)) \wedge f(b) = g(b)$$

Lemma .0.5.

$$\forall f, g.\, \forall a, b.\, a < b \wedge f =_a g \Rightarrow f =_b g$$

Proof. Let $c \in \mathcal{A}$ such that $b <_{\mathcal{A}} c$. Then by transitivity of $<_{\mathcal{A}}$ it follows that $a <_{\mathcal{A}} c$ and thus $c \in \mathrm{dom}(f) \Leftrightarrow c \in \mathrm{dom}(g)$ and $f(c) = g(c)$, as required. □

Corollary .0.6.

$$\begin{aligned}&\forall \mathcal{E}.\, \forall f, f_1, f_2 \in |\textsc{ECtx}|.\, \forall a.\\ &\quad a \in \mathrm{dom}(f) \wedge f_1 =_a f_2 \Rightarrow subst(f(a), f_1) = subst(f(a), f_2)\end{aligned}$$

Proof. By Lemma .0.3 it suffices to prove that

$$b \in \mathrm{dom}(f_1) \Leftrightarrow b \in \mathrm{dom}(f_2) \qquad\qquad f_1(b) = f_2(b)$$

for all $b \in FA(f(a))$. To that end, let $b \in FA(f(a))$. By definition of $|\textsc{ECtx}|$ it follows that $a < b$ and thus by the $f_1 =_a f_2$ assumption it follows that $f_1(b) = f_2(b)$ and $b \in \mathrm{dom}(f_1) \Leftrightarrow b \in \mathrm{dom}(f_2)$, as required. □

Lemma .0.7.

$$\forall f \in |\textsc{ECtx}|.\, \forall a \in \mathcal{A}.\, f =_a (f \setminus \{a\})$$

Proof. Let $b \in \mathcal{A}$ such that $a < b$. Then $a \neq b$ and thus $b \in \mathrm{dom}(f) \Leftrightarrow b \in \mathrm{dom}(f \setminus \{a\})$ and $f(b) = (f \setminus \{a\})(b)$. □

Lemma .0.8.

$$\forall f, g \in |\textsc{ECtx}|.\, g \subseteq f \Rightarrow subst(\mathcal{E}, f) = subst(subst(\mathcal{E}, g), f)$$

Proof. By lexicographic induction on $|g|$ and the size of $\mathcal{E}$.

- Case $\mathcal{E} = a$: if $a \in \mathrm{dom}(g)$ then

$$\begin{aligned}subst(subst(\mathcal{E}, g), f) &= subst(subst(g(a), g \setminus \{a\}), f) \overset{IH}{=} subst(g(a), f)\\ &= subst(f(a), f)\\ &= subst(f(a), f \setminus \{a\})\\ &= subst(\mathcal{E}, f)\end{aligned}$$

  where the second to last equality follows from Corollary .0.6 and Lemma .0.7. If $a \notin \mathrm{dom}(g)$ then

$$subst(subst(\mathcal{E}, g), f) = subst(\mathcal{E}, f)$$

- All the remaining cases follow directly from the induction hypothesis.

□

Lemma .0.9.

$$\forall \mathcal{E} \in EExp.\, \forall f \in |\text{ECTX}|.\, FA(\mathcal{E})\ \textit{defined} \Rightarrow$$
$$FA(subst(\mathcal{E}, f)) = (FA(\mathcal{E}) \setminus \text{dom}(f)) \cup \bigcup \{FA(f(a)) \mid a \in FA(\mathcal{E}) \cap \text{dom}(f)\}$$

Lemma .0.10.

$$\forall \mathcal{E}.\, \forall \kappa.\, \forall f.\, subst(\kappa[\mathcal{E}], f) = subst(\kappa[subst(\mathcal{E}, f)], f)$$

Proof. By induction on the structure of κ.

- Case $\kappa \equiv \bullet$: then $subst(\mathcal{E}, f) = subst(subst(\mathcal{E}, f), f)$ by Lemma .0.8.
- Cse $\kappa \equiv \kappa_1 = \mathcal{E}'$: then

$$\begin{aligned} subst(\kappa_1[\mathcal{E}] = \mathcal{E}', f) &= (subst(\kappa_1[\mathcal{E}], f) = subst(\mathcal{E}', f)) \\ &\overset{IH}{=} (subst(\kappa_1[subst(\mathcal{E}, f)], f) = subst(\mathcal{E}', f)) \\ &= subst(\kappa_1[subst(\mathcal{E}, f)] = \mathcal{E}', f) \\ &= subst(\kappa[subst(\mathcal{E}, f)], f) \end{aligned}$$

- All remaining cases follow directly from the induction hypothesis.

□

Lemma .0.11.

$$\begin{aligned} &\forall \mathcal{E}.\, \forall f.\, \forall j.\, \forall \kappa.\, \forall e \in \text{EXP}.\, \forall k \notin \text{dom}(f). \\ &\quad f(j) = \kappa[e] \wedge j < k \wedge FA(\mathcal{E}, f) = \text{dom}(f) \\ &\qquad \Rightarrow\ subst(\mathcal{E}, f) = subst(\mathcal{E}, f[j \mapsto \kappa[k], k \mapsto e]) \end{aligned}$$

Proof. By lexicographic induction on $|f|$ and the size of $\mathcal{E}$.

- Case $\mathcal{E} = a$: Since $a \in FA(\mathcal{E}, f) = \text{dom}(f)$ and $k \notin \text{dom}(f)$ it follows that $a \neq k$. If $a = j$ then

$$\begin{aligned} &subst(\mathcal{E}, f[j \mapsto \kappa[k], k \mapsto e]) \\ &\quad = subst(\kappa[k], (f \setminus \{j\})[k \mapsto e]) \\ &\quad = subst(subst(\kappa[k], [k \mapsto e]), (f \setminus \{j\})[k \mapsto e]) \\ &\quad = subst(subst(\kappa[subst(k, [k \mapsto e])], [k \mapsto e]), (f \setminus \{j\})[k \mapsto e]) \\ &\quad = subst(subst(\kappa[e], [k \mapsto e]), (f \setminus \{j\})[k \mapsto e]) \\ &\quad = subst(\kappa[e], (f \setminus \{j\})[k \mapsto e]) \\ &\quad = subst(\kappa[e], f \setminus \{j\}) \\ &\quad = subst(\mathcal{E}, f) \end{aligned}$$

and if $a \neq j$ then

$$\begin{aligned} subst(\mathcal{E}, f[j \mapsto \kappa[k], k \mapsto e]) &= subst(f(a), (f \setminus \{a\})[j \mapsto \kappa[k], k \mapsto e]) \\ &\overset{IH}{=} subst(f(a), f \setminus \{a\})) \\ &= subst(\mathcal{E}, f) \end{aligned}$$

- All remaining cases follow directly from induction hypothesis.

□

Lemma .0.12.

$$\begin{aligned} &\forall \mathcal{E}.\, \forall f.\, \forall j, k \in \mathrm{dom}(f).\, \forall \kappa.\, \forall e \in \textsc{Exp}. \\ &\quad f(j) = \kappa[k] \wedge f(k) = e \wedge j \neq k \wedge FA(\mathcal{E}, f) = \mathrm{dom}(f) \\ &\qquad \Rightarrow\ subst(\mathcal{E}, f) = subst(\mathcal{E}, f[j \mapsto \kappa[e], k \mapsto \bot]) \end{aligned}$$

Proof. By lexicographic induction on $|f|$ and $|\mathcal{E}|$.

- Case $\mathcal{E} = a$: If $a = j$ then

$$\begin{aligned} subst(\mathcal{E}, f) &= subst(\kappa[k], f \setminus \{j\}) \\ &= subst(subst(\kappa[k], [k \mapsto e]), f \setminus \{j\}) \\ &= subst(subst(\kappa[subst(k, [k \mapsto e])], [k \mapsto e]), f \setminus \{j\}) \\ &= subst(\kappa[e], f \setminus \{j\}) \\ &= subst(\kappa[e], f[k \mapsto \bot] \setminus \{j\}) \\ &= subst(\mathcal{E}, f[j \mapsto \kappa[e], k \mapsto \bot]) \end{aligned}$$

  where the second to last equality follows from the fact that $k \notin FA(\kappa[e])$.

  If $a = k$ then $\mathrm{dom}(f) = FA(\mathcal{E}, f) = \{a\} \uplus FA(e) = \{a\}$, which is a contradiction, as $k, j \in \mathrm{dom}(f)$ and $k \neq j$.

  Lastly, if $a \neq k$ and $a \neq j$ then

$$\begin{aligned} subst(\mathcal{E}, f) &= subst(f(a), f \setminus \{a\}) \\ &\overset{IH}{=} subst(f(a), f \setminus \{a\}[j \mapsto \kappa[e], k \mapsto \bot]) \\ &= subst(f(a), (f[j \mapsto \kappa[e], k \mapsto \bot]) \setminus \{a\}) \\ &= subst(\mathcal{E}, f[j \mapsto \kappa[e], k \mapsto \bot]) \end{aligned}$$

- All remaining cases follow directly from the induction hypothesis.

□

Lemma .0.13.

$$\forall \kappa.\, \forall k.\, \forall e \in \textsc{Exp}.\, FA(\kappa[k]) = FA(\kappa[e]) \uplus \{k\}$$

Proof. By induction on κ.

- Case $\kappa = \bullet$: then $FA(\kappa[k]) = FA(k) = \{k\} = FA(\kappa[e]) \uplus \{k\}$.
- Case $\kappa = \kappa_1 \| \mathcal{E}$: then

$$\begin{aligned} FA(\kappa[k]) &= FA(\kappa_1[k]) \uplus FA(\mathcal{E}) \\ &\overset{IH}{=} FA(\kappa_1[e]) \uplus \{k\} \uplus FA(\mathcal{E}) = FA(\kappa[e]) \uplus \{k\} \end{aligned}$$

- All remaining cases follows directly from the induction hypothesis.

□

Lemma .0.14.

$$\begin{aligned} &\forall \kappa. \forall f. \forall k \in \mathrm{dom}(f). \forall e \in \textsc{Exp}. \\ &\quad FA(\kappa[k], f) = FA(\kappa[e], f) \uplus \{k\} \uplus FA(f(k), f \setminus \{k\}) \end{aligned}$$

Proof. By induction on the structure of κ.

- Case $\kappa = \bullet$: then

$$\begin{aligned} FA(\kappa[k], f) &= FA(k, f) = \{k\} \uplus FA(f(k), f \setminus \{k\}) \\ &= FA(\kappa[e], f) \uplus \{k\} \uplus FA(f(k), f \setminus \{k\}) \end{aligned}$$

- Case $\kappa = \kappa_1 \| \mathcal{E}$: then

$$\begin{aligned} FA(\kappa[k], f) &= FA(\kappa_1[k]) \uplus FA(\mathcal{E}) \uplus \\ &\qquad \biguplus \{FA(f(a), f \setminus \{a\}) \mid a \in FA(\kappa_1[k]) \uplus FA(\mathcal{E})\} \\ &= FA(\kappa_1[e]) \uplus \{k\} \uplus FA(\mathcal{E}) \uplus FA(f(k), f \setminus \{k\}) \uplus \\ &\qquad \biguplus \{FA(f(a), f \setminus \{a\}) \mid a \in FA(\kappa_1[e]) \uplus FA(\mathcal{E})\} \\ &= FA(\kappa_1[e], f) \uplus \{k\} \uplus FA(f(k), f \setminus \{k\}) \end{aligned}$$

- All remaining cases should follow directly from the induction hypothesis.

□

Lemma .0.15.

$$\begin{aligned} &\forall f. \forall j. \forall \kappa. \forall k \notin \mathrm{dom}(f). \forall e. \\ &\quad f(j) = \kappa[e] \wedge FA(\mathcal{E}, f) = \mathrm{dom}(f) \Rightarrow \\ &\qquad FA(\mathcal{E}, f[j \mapsto \kappa[k], k \mapsto e]) = \mathrm{dom}(f[j \mapsto \kappa[k], k \mapsto e]) \end{aligned}$$

Proof. By lexicographic induction on $|f|$ and the size of $\mathcal{E}$. Let $f' = f[j \mapsto \kappa[k], k \mapsto e]$.

- Case $\mathcal{E} = a$: If $a = k$ then $a \in FA(\mathcal{E}, f) = \mathrm{dom}(f)$ and thus $k \in \mathrm{dom}(f)$, which is a contradiction. If $a = j$ then

$$\begin{aligned} FA(\mathcal{E}, f') &= \{j\} \uplus FA(\kappa[k], (f \setminus \{j\})[k \mapsto e]) \\ &= \{j\} \uplus FA(\kappa[e], (f \setminus \{j\})[k \mapsto e]) \uplus \{k\} \uplus FA(e, f[k \mapsto e]) \\ &= \{j, k\} \uplus FA(\kappa[e], (f \setminus \{j\})[k \mapsto e]) \\ &= \{j, k\} \uplus FA(\kappa[e], f \setminus \{j\}) \\ &= \{k\} \uplus FA(\mathcal{E}, f) \\ &= \{k\} \uplus \mathrm{dom}(f) \\ &= \mathrm{dom}(f') \end{aligned}$$

  Lastly, if $a \neq k$ and $a \neq j$ then

$$\begin{aligned} FA(\mathcal{E}, f') &= \{a\} \uplus FA(f'(a), f' \setminus \{a\}) \\ &= \{a\} \uplus FA(f(a), (f \setminus \{a\})[j \mapsto \kappa[k], k \mapsto e]) \\ &\overset{IH}{=} \{a\} \uplus \mathrm{dom}((f \setminus \{a\})[j \mapsto \kappa[k], k \mapsto e]) \\ &= \{a\} \uplus (\mathrm{dom}(f[j \mapsto \kappa[k], k \mapsto e]) \setminus \{a\}) \\ &= \mathrm{dom}(f') \end{aligned}$$

- All the remaining cases should follow directly from the induction hypothesis.

□

Lemma .0.16.

$$\begin{aligned} &\forall f. \forall j, k \in \mathrm{dom}(f). \forall \kappa. \forall e. \\ &\quad f(j) = \kappa[k] \wedge f(k) = e \wedge j \neq k \wedge FA(\mathcal{E}, f) = \mathrm{dom}(f) \\ &\qquad \Rightarrow FA(\mathcal{E}, f[j \mapsto \kappa[e], k \mapsto \bot]) = \mathrm{dom}(f[j \mapsto \kappa[e], k \mapsto \bot]) \end{aligned}$$

Proof. By lexicographic induction on $|f|$ and the size of $\mathcal{E}$. Let $f' = f[j \mapsto \kappa[e], k \mapsto \bot]$.

- Case $\mathcal{E} = a$: If $a = k$ then $\mathrm{dom}(f) \in FA(\mathcal{E}, f) = \{k\} \uplus FA(e, f \setminus \{k\}) = \{k\}$, which is a contradiction as $k, j \in \mathrm{dom}(f)$ and $k \neq j$. If $a = j$ then

$$\begin{aligned} FA(\mathcal{E}, f') &= \{j\} \uplus FA(\kappa[e], (f \setminus \{j\})[k \mapsto \bot]) \\ &= \{j\} \uplus FA(\kappa[e], f \setminus \{j\}) \\ &= \{j\} \uplus (FA(\kappa[k], f \setminus \{j\}) \setminus \{k\}) \\ &= FA(\mathcal{E}, f) \setminus \{k\} \\ &= \mathrm{dom}(f) \setminus \{k\} \\ &= \mathrm{dom}(f') \end{aligned}$$

Lastly, if $a \neq k$ and $a \neq j$ then

$$\begin{aligned} FA(\mathcal{E}, f') &= \{a\} \uplus FA(f'(a), f' \setminus \{a\}) \\ &= \{a\} \uplus FA(f(a), (f \setminus \{a\})[j \mapsto \kappa[e], k \mapsto \bot]) \\ &\overset{IH}{=} \{a\} \uplus \mathrm{dom}((f \setminus \{a\})[j \mapsto \kappa[e], k \mapsto \bot]) \\ &= \{a\} \uplus (\mathrm{dom}(f[j \mapsto \kappa[e], k \mapsto \bot]) \setminus \{a\}) \\ &= \mathrm{dom}(f') \end{aligned}$$

- All the remaining cases should follow directly from the induction hypothesis.

□

Lemma .0.17.

$$\forall f. \forall \mathcal{E}. \forall a \in \mathrm{dom}(f). a \notin FA(\mathcal{E}, f) \Rightarrow FA(\mathcal{E}, f) = FA(\mathcal{E}, f[a \mapsto \bot])$$

Proof. By lexicographic induction on $|f|$ and $|\mathcal{E}|$.

- Case $\mathcal{E} = b$: since $a \notin FA(\mathcal{E}, f) = \{b\} \uplus FA(f(b), f \setminus \{b\})$ it follows that $a \neq b$. We thus have,

$$\begin{aligned} FA(\mathcal{E}, f) &= \{b\} \uplus FA(f(b), f \setminus \{b\}) \\ &\overset{IH}{=} \{b\} \uplus FA(f(b), (f \setminus \{b\})[a \mapsto \bot]) \\ &= \{b\} \uplus FA(f(b), (f[a \mapsto \bot]) \setminus \{b\}) \\ &= FA(\mathcal{E}, f[a \mapsto \bot]) \end{aligned}$$

- All the remaining cases follow directly from the induction hypothesis.

□

Lemma .0.18.

$$\forall f. \forall \mathcal{E}. \forall a \in \mathrm{dom}(f). a \notin FA(\mathcal{E}, f) \Rightarrow subst(\mathcal{E}, f) = subst(\mathcal{E}, f[a \mapsto \bot])$$

Proof. By lexicographic induction on $|f|$ and $|\mathcal{E}|$.

- Case $\mathcal{E} = b$: since $a \notin FA(\mathcal{E}, f) = \{b\} \uplus FA(f(b), f \setminus \{b\})$ it follows that $a \neq b$. We thus have,

$$\begin{aligned} subst(\mathcal{E}, f) &= subst(f(b), f \setminus \{b\}) \\ &\overset{IH}{=} subst(f(b), (f \setminus \{b\})[a \mapsto \bot]) \\ &= subst(f(b), (f[a \mapsto \bot] \setminus \{b\})) \\ &= subst(\mathcal{E}, f[a \mapsto \bot]) \end{aligned}$$

- All the remaining cases follow directly from the induction hypothesis.

□

Lemma .0.19.

$$\begin{aligned}&\forall \mathcal{E}.\, \forall f.\, \forall j \in \mathrm{dom}(f).\, FA(\mathcal{E}, f) = \mathrm{dom}(f) \wedge FA(f(j)) = \emptyset \\ &\quad \Rightarrow\ \exists K.\, \forall e \in \textsc{Exp}.\, subst(\mathcal{E}, f[j \mapsto e]) = K[e]\end{aligned}$$

Proof. By lexicographic induction on $|f|$ and $|\mathcal{E}|$.

- Case $\mathcal{E} = a$: if $a = j$ then $\mathrm{dom}(f) = FA(\mathcal{E}, f) = FA(f(a)) \uplus \{j\} = \{j\}$. We thus take $K = \bullet$. Then, for every $e \in \textsc{Exp}$ we have

  $$subst(\mathcal{E}, f[j \mapsto e]) = subst(e, []) = e = K[e]$$

  If $a \neq j$ then $FA(f(a), f \setminus \{a\}) = \mathrm{dom}(f \setminus \{a\})$ and by the induction hypothesis, there exists a K such that $subst(f(a), (f \setminus \{a\})[j \mapsto e]) = K[e]$. We simply pick this K:

  $$subst(\mathcal{E}, f[j \mapsto e]) = subst(f(a), (f \setminus \{a\})[j \mapsto e]) = K[e]$$

- Case $\mathcal{E} = \mathcal{E}_1\ \mathcal{E}_2$: we know that $j \in \mathrm{dom}(f) = FA(\mathcal{E}, f) = FA(\mathcal{E}_1, f) \uplus FA(\mathcal{E}_2, f)$. By Lemma .0.17 it follows that $FA(\mathcal{E}_1, f) = FA(\mathcal{E}_1, f \setminus FA(\mathcal{E}_2, f))$ and $FA(\mathcal{E}_2, f) = FA(\mathcal{E}_2, f \setminus FA(\mathcal{E}_1, f))$ and more importantly,

  $$\begin{aligned}FA(\mathcal{E}_1, f \setminus FA(\mathcal{E}_2, f)) &= \mathrm{dom}(f \setminus FA(\mathcal{E}_2, f)) \\ FA(\mathcal{E}_2, f \setminus FA(\mathcal{E}_1, f)) &= \mathrm{dom}(f \setminus FA(\mathcal{E}_1, f))\end{aligned}$$

  If $j \in FA(\mathcal{E}_1, f)$ then by the induction hypothesis, there exists a K such that

  $$subst(\mathcal{E}_1, (f \setminus FA(\mathcal{E}_2, f))[j \mapsto e]) = K[e]$$

  for all expressions e. We thus simply pick $K\ subst(\mathcal{E}_2, f)$ as our context, such that

  $$\begin{aligned}subst(\mathcal{E}, f[j \mapsto e]) &= subst(\mathcal{E}_1, f[j \mapsto e])\ subst(\mathcal{E}_2, f[j \mapsto e]) \\ &= subst(\mathcal{E}_1, (f \setminus FA(\mathcal{E}_2, f))[j \mapsto e])\ subst(\mathcal{E}_2, f) \\ &= K[e]\ subst(\mathcal{E}_2, f)\end{aligned}$$

  for all expressions e. Here the second equality follows by Lemma .0.18.

  The case of $j \in FA(\mathcal{E}_2, f)$ is symmetric.

- All other cases follow a similar pattern: on binary expression formers, do a case-analysis on which sub-expression j "appears" in and appeal to the induction hypothesis for that sub-expression.

□

Definition .0.20.

$$j \overset{\zeta}{\Rightarrow}_S B \triangleq \boxed{\circ[j \mapsto B] : \textsc{AuthECtx}}^{\textsc{Exp}(\zeta)}$$

$$mctx(e,\zeta) \triangleq \exists f \in |\textsc{ECtx}|. \boxed{\bullet f : \textsc{AuthECtx}}^{\textsc{Exp}(\zeta)} * subst(0,f) = e * FA(0,f) = \mathrm{dom}(f)$$

Lemma .0.21.

$$mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S \kappa[e'] \Rrightarrow \exists k.\, mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e'$$

Proof.

$$\begin{aligned}
&mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S \kappa[e'] \\
&= \exists f.\, j \overset{\zeta}{\Rightarrow}_S \kappa[e'] * \boxed{\bullet f}^{\zeta} * subst(0,f) = e * FA(0,f) = \mathrm{dom}(f) \\
&\Rightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[e'] * \boxed{\bullet f}^{\zeta} * subst(0,f') = e * FA(0,f) = \mathrm{dom}(f) \\
&\Rightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[e'] * \boxed{\bullet f}^{\zeta} * subst(0,f') = e * FA(0,f') = \mathrm{dom}(f') \\
&\Rrightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' * \boxed{\bullet f'}^{\zeta} * subst(0,f') = e * FA(0,f') = \mathrm{dom}(f') \\
&\Rightarrow \exists k.\, j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' * mctx(e,\zeta)
\end{aligned}$$

where $f' = f[j \mapsto \kappa[k], k \mapsto e']$, the first implication follows by Lemma .0.11 and the second implication by Lemma .0.15. □

Lemma .0.22.

$$mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' \Rrightarrow mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S \kappa[e']$$

Proof.

$$\begin{aligned}
&mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' \\
&= \exists f.\, j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' * \boxed{\bullet f}^{\zeta} * subst(0,f) = e * FA(0,f) = \mathrm{dom}(f) \\
&\Rightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' * \boxed{\bullet f}^{\zeta} * subst(0,f') = e * FA(0,f) = \mathrm{dom}(f) \\
&\Rightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[k] * k \overset{\zeta}{\Rightarrow}_S e' * \boxed{\bullet f}^{\zeta} * subst(0,f') = e * FA(0,f') = \mathrm{dom}(f') \\
&\Rrightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[e'] * \boxed{\bullet f'}^{\zeta} * subst(0,f') = e * FA(0,f') = \mathrm{dom}(f') \\
&\Rightarrow j \overset{\zeta}{\Rightarrow}_S \kappa[e'] * mctx(e,\zeta)
\end{aligned}$$

where $f' = f[j \mapsto \kappa[e'], k \mapsto \bot]$, the first implication follows by Lemma .0.12 and the second implication by Lemma .0.16. □

Lemma .0.23.

$$mctx(e,\zeta) * 0 \overset{\zeta}{\Rightarrow}_S e' \Rightarrow mctx(e,\zeta) * 0 \overset{\zeta}{\Rightarrow}_S e' * e = e'$$

Proof. By unfolding the syntactic sugar, it follows that $subst(e', f) = e$ and since $FA(e') = \emptyset$ we have $e = e'$ as required. □

Lemma .0.24.

$$\begin{aligned}
&\forall e, e', e_1, e_1'. \forall h, h'. \forall j. \\
&\quad mctx(e,\zeta) * j \overset{\zeta}{\Rightarrow}_S e_1 * (h; e_1 \to h'; e_1') \Rrightarrow \\
&\quad\quad \exists e'. mctx(e',\zeta) * j \overset{\zeta}{\Rightarrow}_S e_1' * (h; e \to h'; e')
\end{aligned}$$

Proof. If $j = 0$ then it follows by Lemma .0.23 that $e = e_1$ and the conclusion thus follows easily by taking $e' = e_1'$.

Otherwise, $j \neq 0$ and by unfolding the syntactic sugar there exists an f such that

$$\boxed{\bullet f}^{\zeta} * e = subst(0, f) * FA(0, f) = \mathrm{dom}(f) * \boxed{\circ [j \mapsto e_1]}^{\zeta} * (h; e_1 \to h'; e_1')$$

By Lemma .0.19 there exists a K such that

$$subst(0, f[j \mapsto e'']) = K[e'']$$

for all expressions e''. Hence, in particular, $e = subst(0, f[j \mapsto e_1]) = K[e_1]$. We thus have

$$\begin{aligned}
&\boxed{\bullet f}^{\zeta} * e = subst(0, f) * FA(0, f) = \mathrm{dom}(f) * \boxed{\circ [j \mapsto e_1]}^{\zeta} * (h; e_1 \to h'; e_1') \\
&\Rightarrow \boxed{\bullet f}^{\zeta} * K[e_1'] = subst(0, f[j \mapsto e_1']) * FA(0, f[j \mapsto e_1']) = \mathrm{dom}(f[j \mapsto e_1']) * \\
&\quad \boxed{\circ [j \mapsto e_1]}^{\zeta} * (h; K[e_1] \to h'; K[e_1']) \\
&\Rrightarrow \boxed{\bullet f[j \mapsto e_1']}^{\zeta} * K[e_1'] = subst(0, f[j \mapsto e_1']) * \\
&\quad FA(0, f[j \mapsto e_1']) = \mathrm{dom}(f[j \mapsto e_1']) * \boxed{\circ [j \mapsto e_1']}^{\zeta} * \\
&\quad (h; K[e_1] \to h'; K[e_1']) \\
&\Rightarrow mctx(K[e_1'],\zeta) * j \overset{\zeta}{\Rightarrow}_S e_1' * (h; e \to h'; K[e_1'],)
\end{aligned}$$

□

Standard Iris Monoids

$$\begin{aligned}
\textsc{AHeap} &\triangleq \textsc{AuthFpFun}(\textsc{Loc}, \textsc{Val}) \\
\textsc{Sr} &\triangleq \textsc{Frac}\{*\} \\
\textsc{Reg} &\triangleq \textsc{FpFun}(\mathcal{RN}, \textsc{Frac}X + (\{A \in \mathcal{P}(X) \mid |A| = 2\} \times \textsc{Heap})) \\
&\quad \text{where } X \triangleq \mathsf{list}\ Name \\
\textsc{AFHeap} &\triangleq \textsc{AuthFpFun}(\textsc{Loc}, \textsc{FracVal}) \\
\textsc{EfReg} &\triangleq \textsc{FpFun}(\mathbb{N}, \textsc{Frac}\{*\}) \\
\textsc{EfRegLoc} &\triangleq \textsc{FpFun}(\textsc{Loc}, \textsc{Ex}\{*\}) \\
\textsc{AllocHeap} &\triangleq \textsc{Frac}\mathcal{P}(\textsc{Loc}) \times \mathcal{P}(\textsc{Loc})
\end{aligned}$$

Disjoint Monoid

Assume a countably infinite set X, define:

$$\textsc{Disjoint} \triangleq (\mathcal{P}(X), \circ, \emptyset)$$

where

$$x \circ y \triangleq x \cup y \text{ if } x \# y$$

Syntactic Sugar

$\mathbf{LR_{ML}}$

$$\begin{aligned}
heap(h) &\triangleq \boxed{\bullet h : \textsc{AHeap}}^{\pi_1(\gamma)} \\
l \mapsto v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AHeap}}^{\pi_1(\gamma)}
\end{aligned}$$

$\mathbf{LR}_{\mathbf{Eff}}$

$$
\begin{aligned}
heap(h) &\triangleq \boxed{\bullet h : \textsc{AHeap}}^{\pi_1(\gamma)} \\
l \mapsto v &\triangleq \boxed{\circ[l \mapsto v] : \textsc{AHeap}}^{\pi_1(\gamma)} \\
[\textsc{Rd}]^{\pi}_{r} &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_2(\gamma)} \\
[\textsc{Wr}]^{\pi}_{r} &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_3(\gamma)} \\
[\textsc{Al}]^{\pi}_{r} &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_4(\gamma)} \\
rheap(h, r) &\triangleq \boxed{\bullet \widehat{h} : \textsc{AFHeap}}^{R(r)} \\
x \overset{\pi}{\hookrightarrow}_r v &\triangleq \boxed{\circ[l \mapsto v] : \textsc{AFHeap}}^{R(r)} \\
[\textsc{Rd}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{Rd}(r)} \\
[\textsc{NoRd}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{No}(r)} \\
[\textsc{Wr}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{Wr}(r)} \\
[\textsc{Al}(h)]^{\pi}_{r} &\triangleq \boxed{(\pi, \mathrm{dom}(h)) : \textsc{AllocHeap}}^{\textsc{Al}(r)}
\end{aligned}
$$

$\mathbf{LR_{Bin}}$

$$
\begin{aligned}
heap_I(h) &\triangleq \boxed{\bullet h : \textsc{AHeap}}^{\pi_1(\gamma)} \\
l \mapsto_I v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AHeap}}^{\pi_1(\gamma)} \\
heap_S(h) &\triangleq \boxed{\bullet h : \textsc{AHeap}}^{\pi_2(\gamma)} \\
l \mapsto_S v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AHeap}}^{\pi_2(\gamma)} \\
[\textsc{Rd}]^\pi_r &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_5(\gamma)} \\
[\textsc{Wr}]^\pi_r &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_6(\gamma)} \\
[\textsc{Al}]^\pi_r &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_7(\gamma)} \\
mctx(f) &\triangleq \boxed{\bullet f : \textsc{AuthECtx}}^{\pi_8(\gamma)} \\
j \Rightarrow_S e &\triangleq \boxed{\circ [j \mapsto e] : \textsc{AuthECtx}}^{\pi_8(\gamma)}
\end{aligned}
$$

$$
\begin{aligned}
rheap_X(h, r) &\triangleq \boxed{\bullet \widehat{h} : \textsc{AFHeap}}^{X(r)} \\
x \overset{\pi}{\hookrightarrow}_{X,r} v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AFHeap}}^{X(r)} \\
[\textsc{Rd}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{Rd}(r)} \\
[\textsc{NoRd}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{No}(r)} \\
[\textsc{Wr}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{Wr}(r)} \\
[\textsc{Al}(h_1, h_2)]^\pi_r &\triangleq \boxed{(\pi, (\mathrm{dom}(h_1), \mathrm{dom}(h_2))) : \textsc{AllocHeap}}^{\textsc{Al}(r)}
\end{aligned}
$$

$\mathbf{LR_{PAR}}$

$$
\begin{aligned}
heap_I(h) &\triangleq \boxed{\bullet h : \textsc{AHeap}}^{\pi_1(\gamma)} \\
l \mapsto_I v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AHeap}}^{\pi_1(\gamma)} \\
[\textsc{Mu}(r, \{\zeta\})]^\pi &\triangleq \boxed{[r \mapsto (\pi, \mathsf{inj}_1\ \zeta)] : \textsc{Reg}}^{\pi_2(\gamma)} \\
[\textsc{Im}(r, S, h)]^\pi &\triangleq \boxed{[r \mapsto (\pi, \mathsf{inj}_2\ (S, h))] : \textsc{Reg}}^{\pi_2(\gamma)} \\
[Y]_H &\triangleq \boxed{Y : \textsc{Disjoint}}^{\pi_3(\gamma)} \\
[\textsc{Rd}]^\pi_r &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_4(\gamma)} \\
[\textsc{Wr}]^\pi_r &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_5(\gamma)} \\
[\textsc{Al}]^\pi_r &\triangleq \boxed{[r \mapsto (\pi, *)] : \textsc{EfReg}}^{\pi_6(\gamma)}
\end{aligned}
$$

$$
\begin{aligned}
heap_S(h, \zeta) &\triangleq \boxed{\bullet h : \textsc{AHeap}}^{\pi_1(\zeta)} \\
l \mapsto^\zeta_S v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AHeap}}^{\pi_1(\zeta)} \\
mctx(f) &\triangleq \boxed{\bullet f : \textsc{AuthECtx}}^{\pi_2(\zeta)} \\
j \Rightarrow^\zeta_S e &\triangleq \boxed{\circ [j \mapsto e] : \textsc{AuthECtx}}^{\pi_2(\zeta)} \\
[\textsc{Sr}]^\pi_\zeta &\triangleq \boxed{(\pi, *) : \textsc{Sr}}^{\pi_3(\zeta)}
\end{aligned}
$$

$$
\begin{aligned}
rheap_X(h, r) &\triangleq \boxed{\bullet \widehat{h} : \textsc{AFHeap}}^{X(r)} \\
x \hookrightarrow^\pi_{X,r} v &\triangleq \boxed{\circ [l \mapsto v] : \textsc{AFHeap}}^{X(r)} \\
[\textsc{Rd}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{Rd}(r)} \\
[\textsc{NoRd}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{No}(r)} \\
[\textsc{Wr}(x)]_r &\triangleq \boxed{[x \mapsto *] : \textsc{EfRegLoc}}^{\textsc{Wr}(r)} \\
[\textsc{Al}(h_1, h_2)]^\pi_r &\triangleq \boxed{(\pi, (\mathrm{dom}(h_1), \mathrm{dom}(h_2))) : \textsc{AllocHeap}}^{\textsc{Al}(r)}
\end{aligned}
$$

The function $\widehat{\ }$ embeds a partial finite function into a full fractional partial finite function, formally, it is pairwise applied where each map is computed as so:

$$\widehat{x \mapsto v} = x \mapsto (1, v)$$

Utility functions for invariant names

Throughout the entire paper we assume a constant invariant name Hp and functions Sp, Rg and Rf that maps simulation identifiers, region identifiers

and locations into Iris names respectively. We assume each function is injective, that the images of each pair of functions is disjoint and does not contain Hp.

The $\mathbf{LR_{ML}}$ relation

We assume a list of monoid-names γ to be defined globally.

$$\textsc{Heap} \triangleq \exists h.\ heap(h) * \lfloor h \rfloor$$
$$\textsc{Ref}(\phi, x) \triangleq \exists v.\ x \mapsto v * \phi(v)$$

$$\begin{aligned}
[\![\mathbf{1}]\!] &\triangleq \lambda x.\ x = () \\
[\![\mathbf{B}]\!] &\triangleq \lambda x.\ x \in \{\mathbf{true}, \mathbf{false}\} \\
[\![\mathbf{int}]\!] &\triangleq \lambda x.\ x \in \mathbb{N} \\
[\![\tau_1 \times \tau_2]\!] &\triangleq \lambda x.\ \exists y_1, y_2.\ x = (y_1, y_2) \ \wedge \triangleright\, y_1 \in [\![\tau_1]\!] \wedge \triangleright\, y_2, \in [\![\tau_2]\!] \\
[\![\tau_1 + \tau_2]\!] &\triangleq \lambda x.\ (\triangleright \exists y \in [\![\tau_1]\!].\ x = \mathbf{inj}_1\ y) \ \vee (\triangleright \exists y \in [\![\tau_2]\!].\ x = \mathbf{inj}_2\ y) \\
[\![\tau_1 \to \tau_2]\!] &\triangleq \lambda x.\ \Box \forall y.\ (\triangleright\, y \in [\![\tau_1]\!]) \ \Rightarrow \mathcal{E}([\![\tau_2]\!])(x\ y) \\
[\![\mathbf{ref}\ \tau]\!] &\triangleq \lambda x.\ \boxed{\textsc{Ref}([\![\tau]\!], x)}^{\mathrm{Rf}(x)}
\end{aligned}$$

$$\mathcal{E}(\phi) \triangleq \lambda x.\ \left\{\boxed{\textsc{Heap}}^{\mathrm{Hp}}\right\} x \{v.\ \phi(v)\}_{\top}$$

Logical relatedness

$$\overline{x : \tau} \models_{\mathrm{ML}} e : \tau \triangleq \vdash_{\mathrm{Iris}} \forall \overline{x'}.\ \overline{[\![\tau]\!]}(\overline{x'}) \implies \mathcal{E}([\![\tau]\!])(e[\overline{x'/x}])$$

Theorem .0.25 (Fundamental Theorem)**.** *If* $\Pi \mid \Delta \mid \Gamma \vdash e : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \models_{\mathrm{ML}} e : \tau, \varepsilon$

Proof. Proof omitted. □

The $\mathbf{LR_{Eff}}$ relation

We assume a list of monoid-names γ to be defined globally.

$$\textsc{Heap} \triangleq \exists h.\ heap(h) * \lfloor h \rfloor$$
$$\textsc{Ref}(r, \phi, x) \triangleq \exists v.\ x \overset{\frac{1}{2}}{\hookrightarrow}_r v * effs(r, \phi, x, v)$$
$$\textsc{Reg}(r) \triangleq locs(r) * tokens(r)$$

where

$$M : \mathcal{RV} \xrightarrow{\text{fin}} MonoidName \text{ list}$$

$$\begin{aligned}
effs(r,\phi,x,v) &\triangleq ([\textsc{Wr}(x)]_r \vee x \overset{\frac{1}{2}}{\hookrightarrow}_r v) * ([\textsc{Rd}(x)]_r \vee (\phi(v) * [\textsc{NoRd}(x)]_r)) \\
locs(r) &\triangleq \exists h.\ rheap(h,r) * alloc(h,r) * \circledast_{(l,v)\in h} l \mapsto v * \\
&\qquad \circledast_{\{x \mid x \in Loc \setminus \mathrm{dom}(h)\}} [\textsc{NoRd}(x)]_r \\
toks(r) &\triangleq ([\textsc{Wr}]_r^{\pi_{wr}} \vee \circledast_{x\in Loc}[\textsc{Wr}(x)]_r) * ([\textsc{Rd}]_r^{\pi_{rd}} \vee \circledast_{x\in Loc}[\textsc{Rd}(x)]_r) \\
alloc(h,r) &\triangleq ([\textsc{Al}(r)]_1 * [\textsc{Al}(h)]_r^{\frac{1}{2}}) \vee [\textsc{Al}(h)]_r^1
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{1}]\!]^M &\triangleq \lambda x.\ x = () \\
[\![\mathbf{B}]\!]^M &\triangleq \lambda x.\ x \in \{\mathbf{true}, \mathbf{false}\} \\
[\![\mathbf{int}]\!]^M &\triangleq \lambda x.\ x \in \mathbb{N} \\
[\![\tau_1 \times \tau_2]\!]^M &\triangleq \lambda x.\ \exists y_1, y_2.\ x = (y_1, y_2)\ \wedge \triangleright y_1 \in [\![\tau_1]\!]^M \wedge \triangleright y_2, \in [\![\tau_2]\!]^M \\
[\![\tau_1 + \tau_2]\!]^M &\triangleq \lambda x.\ (\triangleright \exists y \in [\![\tau_1]\!]^M.\ x = \mathbf{inj}_1\ y)\ \vee (\triangleright \exists y \in [\![\tau_2]\!]^M.\ x = \mathbf{inj}_2\ y) \\
[\![\tau_1 \to_\varepsilon^{\Pi,\Lambda} \tau_2]\!]^M &\triangleq \lambda x.\ \Box \forall y.\ (\triangleright y \in [\![\tau_1]\!]^M)\ \Rightarrow \mathcal{E}_{\varepsilon,M}^{\Pi;\Lambda}([\![\tau_2]\!]^M)(x\ y)) \\
[\![\mathbf{ref}_\rho\ \tau]\!]^M &\triangleq \lambda x.\ \boxed{\textsc{Ref}(M(\rho), [\![\tau]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))}
\end{aligned}$$

$$\begin{aligned}
P_{toks}(\rho, r, \pi, \varepsilon) &\triangleq (\rho \notin \mathsf{rds}\ \varepsilon \vee [\textsc{Rd}]_r^\pi) * (\rho \notin \mathsf{wrs}\ \varepsilon \vee [\textsc{Wr}]_r^\pi) * \\
&\qquad (\rho \notin \mathsf{als}\ \varepsilon \vee [\textsc{Al}]_r^\pi) \\
P_{reg}(R, g, \varepsilon, M) &\triangleq \circledast_{\rho\in R} P_{toks}(\rho, M(\rho), g(\rho), \varepsilon) * \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}_{\varepsilon,M}^{\Pi;\Lambda}(\phi) \triangleq \lambda x.\ \forall g \in \Pi \to Perm.\ & \\
& \{P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M) * P_{reg}(\Pi, g, \varepsilon, M)\} \\
\boxed{\textsc{Heap}}^{\textsc{Hp}} \vdash\ & x \\
& \{v.\ \phi(v) * P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M) * P_{reg}(\Pi, g, \varepsilon, M)\}_\top
\end{aligned}$$

Logical relatedness

$$\begin{aligned}
&\Pi \mid \Lambda \mid \overline{x:\tau} \models_{\textsc{Eff}} e : \tau, \varepsilon \triangleq \\
&\quad \vdash_{\textsc{Iris}} \forall M. \forall \overline{x'}. \overline{[\![\tau]\!]^M}(\overline{x'}) \implies \mathcal{E}_{\varepsilon,M}^{\Pi;\Lambda}([\![\tau]\!]^M)(e[\overline{x'/x}])
\end{aligned}$$

Theorem .0.26 (Fundamental Theorem)**.** *If* $\Pi \mid \Delta \mid \Gamma \vdash e : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \models_{\textsc{Eff}} e : \tau, \varepsilon$

Proof. Proof omitted. □

Example: Type violating assignments

The code below illustrates the possibility to temporarily break the type-constraints for references in private regions.

$$x := ();x := \mathsf{True}$$

The above example clearly violates the type of the parameter x, however, we would still like to show:

$$\cdot \mid \cdot \mid \mathbf{ref}_\rho\ \mathsf{B} \vdash x := ();x := \mathsf{True} : \mathbf{1}, \left\{wr_\rho, rd_\rho\right\}$$

which means we would have to show for $M = M'[\rho \mapsto r]$:

$$\mathcal{E}^{\cdot;\rho}_{\{wr_\rho, rd_\rho\},M}([\![\mathbf{1}]\!]^M)(x := ();x := \mathsf{True})$$

We define the following evaluation context:

$$K^1 \triangleq [];x := \mathsf{True}$$

Lemma .0.27.

$$\forall r.\ \triangleright \textsc{Reg}(r) \Longleftrightarrow \textsc{Reg}(r)$$

Proof. $\triangleright$ can be removed by VSTimeless since ghost resources are timeless. □

Lemma .0.28.

$$\forall r, \phi, x.\ \triangleright \textsc{Ref}(r, \phi, x) \Longleftrightarrow \textsc{Ref}(r, \triangleright \phi, x)$$

Lemma .0.29 (Trade write tokens)**.**

$$\forall h, r.\ tokens(h, 1, 1, r) * [\textsc{Wr}]^1_r \Longleftrightarrow tokens(h, 1, 1, r) * \circledast_{x \in Loc} [\textsc{Wr}(x)]_r$$

Lemma .0.30 (Trade read tokens)**.**

$$\forall h, r.\ tokens(h, 1, 1, r) * [\textsc{Rd}]^1_r \Longleftrightarrow tokens(h, 1, 1, r) * \circledast_{x \in Loc} [\textsc{Rd}(x)]_r$$

Lemma .0.31 (Trade region points-to)**.**

$$\forall r, \phi, x, v.\ effs(r, \phi, x, v) * [\textsc{Wr}(x)]_r \Leftrightarrow effs(r, \phi, x, v) * x \overset{\frac{1}{2}}{\hookrightarrow}_r v$$

Lemma .0.32 (Trade Read for NoRead)**.**

$$\forall r, \phi, x, v.\ effs(r, \phi, x, v) * [\textsc{Rd}(x)]_r \Leftrightarrow effs(r, \phi, x, v) * \phi(v) * [\textsc{NoRd}(x)]_r$$

Lemma .0.33 (Region heap has mapping)**.**

$$\forall h, x, v, \pi, r.\ locs(h, r) * x \overset{\pi}{\hookrightarrow}_r v \Rightarrow \exists h'.\ h = h'[x \mapsto v]$$

Proof. By owning an authorative fragment $x \overset{\pi}{\hookrightarrow}_r v$ it must be that for $regheap(\hat{h}, r)$, $\hat{h}$ contains $[x \mapsto v]$ since this is the corresponding authorative element. Since the hat function is just an injection from a partial map to one with a full fragment, there exists some h' such that $h = h'[x \mapsto v]$. □

Lemma .0.34 (Obtain points-to)**.**

$$\forall h, h', r, x, v.\ h = h'[x \mapsto v] * locs(h, r) \Leftrightarrow regheap(\hat{h}, r) * alloc(h, r) * \circledast_{(l,v') \in h'} l \mapsto v' * x \mapsto v$$

Lemma .0.35 (Update concrete heap)**.**

$$\forall x, v.\ \boxed{\textsc{Heap}}^{\textsc{Hp}} \vdash \begin{array}{l} \{x \mapsto -\} \\ \quad x := v \\ \{v'.\ v' = () * x \mapsto v\} \end{array}$$

Proof.

Context: $x, v, \boxed{\textsc{Heap}}^{\textsc{Hp}}$

$\{x \mapsto -\}_{\{\textsc{Hp}\}}$

Open Hp:

- $\{\triangleright \textsc{Heap} * x \mapsto -\}$
- $\{\textsc{Heap} * x \mapsto -\}$
- $\{\exists h.\ heap(h[x \mapsto -], \gamma) * \lfloor h[x \mapsto -] \rfloor * x \mapsto -\}$
- $x := v$
- $\{v'.\ v' = () * \exists h.\ heap(h[x \mapsto v], \gamma) * \lfloor h[x \mapsto v] \rfloor * x \mapsto v\}$
- $\{v'.\ v' = () * \textsc{Heap} * x \mapsto v\}$

$\{v'.\ v' = () * x \mapsto v\}_{\{\textsc{Hp}\}}$

□

Lemma .0.36 (Make type-violating assignment)**.**

$$\forall r, x, v, \phi.\ \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)} \vdash \begin{array}{l} \{[\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\} \\ \quad x := v \\ \{v'.\ v' = () * [\textsc{Wr}]^1_r * \circledast_{x' \in Loc \setminus \{x\}} [\textsc{Rd}(x')]_r * [\textsc{NoRd}(x)]_r\} \end{array}$$

Proof.

Context: $r, x, v, \phi, \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(x,\phi,x)}^{\textsc{Rf}(x)}$

$\left\{[\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\right\}_{\{\textsc{Hp},\textsc{Rg}(r),\textsc{Rf}(x)\}}$

Open $\textsc{Rg}(r), \textsc{Rf}(x)$

$\left\{\triangleright\,\textsc{Reg}(r) * \triangleright\,\textsc{Ref}(r,\phi,x) * [\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\right\}_{\{\textsc{Hp}\}}$

By Lemma .0.27 and Lemma .0.28

$\left\{\textsc{Reg}(r) * \textsc{Ref}(r,\triangleright\,\phi,x) * [\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\right\}_{\{\textsc{Hp}\}}$

By Lemma .0.29 and Lemma .0.30

$$\left\{\begin{array}{l}\exists h.\ locs(h,r) * tokens(h,1,1,r) * \textsc{Ref}(r,\triangleright\,\phi,x) * \\ \circledast_{x'\in Loc\setminus\{x\}}([\textsc{Wr}(x')]_r * [\textsc{Rd}(x')]_r) * [\textsc{Wr}(x)]_r * [\textsc{Rd}(x)]_r\end{array}\right\}_{\{Hp\}}$$

Frame

$\{\exists h.\ locs(h,r) * \textsc{Ref}(r,\triangleright\,\phi,x) * [\textsc{Wr}(x)]_r * [\textsc{Rd}(x)]_r\}_{\{\textsc{Hp}\}}$

$$\left\{\exists h.\ locs(h,r) * x \overset{\frac{1}{2}}{\hookrightarrow}_r - * effs(r,\phi,x,-) * [\textsc{Wr}(x)]_r * [\textsc{Rd}(x)]_r\right\}_{\{\textsc{Hp}\}}$$

By Lemma .0.31, Lemma .0.32 and Lemma .0.33

$$\left\{\exists h.\ locs(h[x \mapsto -],r) * x \overset{1}{\hookrightarrow}_r - * effs(r,\phi,x,-) * [\textsc{NoRd}(x)]_r\right\}_{\{\textsc{Hp}\}}$$

By Lemma .0.34

$$\left\{\begin{array}{l}\exists h.\ regheap(h[x \overset{\wedge}{\mapsto} -],r) * alloc(h[x \mapsto -],r) * \circledast_{(l,w)\in h} l \mapsto w * \\ x \mapsto - * x \overset{1}{\hookrightarrow}_r - * effs(r,\phi,x,-) * [\textsc{NoRd}(x)]_r\end{array}\right\}_{\{\textsc{Hp}\}}$$

Frame

$\{x \mapsto -\}_{\{\textsc{Hp}\}}$

$x := v$

$\{v'.\ v' = () * x \mapsto -\}_{\{\textsc{Hp}\}}$ By Lemma .0.35

$$\left\{\begin{array}{l}v'.\ v' = () * \exists h.\ regheap(h[x \overset{\wedge}{\mapsto} -],r) * alloc(h[x \mapsto -],r) * \\ \circledast_{(l,w)\in h} l \mapsto w * x \mapsto v * x \overset{1}{\hookrightarrow}_r - * effs(r,\phi,x,-) * [\textsc{NoRd}(x)]_r\end{array}\right\}_{\{\textsc{Hp}\}}$$

Updated region points-to by having full fraction and having both the full and the fragmental authorative parts by AFHeapUpd.

$$\left\{\begin{array}{l}v'.\ v' = () * \exists h.\ regheap(h[x \overset{\wedge}{\mapsto} v],r) * alloc(h[x \mapsto -],r) * \\ \circledast_{(l,w)\in h} l \mapsto w * x \mapsto v * x \overset{1}{\hookrightarrow}_r v * effs(r,\phi,x,v) * [\textsc{NoRd}(x)]_r\end{array}\right\}_{\{\textsc{Hp}\}}$$

$$\left\{\begin{array}{l}v'.\ v' = () * \exists h.\ locs(h,r) * x \overset{\frac{1}{2}}{\hookrightarrow}_r v * x \overset{\frac{1}{2}}{\hookrightarrow}_r v * effs(r,\phi,x,v) * \\ [\textsc{NoRd}(x)]_r\end{array}\right\}_{\{\textsc{Hp}\}}$$ By Lemma .0.31

$\{v'.\ v' = () * \exists h.\ locs(h,r) * \textsc{Ref}(r,\phi,x) * [\textsc{Wr}(x)]_r * [\textsc{NoRd}(x)]_r\}_{\{\textsc{Hp}\}}$

$$\left\{\begin{array}{l}v'.\ v' = () * \exists h.\ locs(h,r) * tokens(h,1,1,r) * \textsc{Ref}(r,\phi,x) * \\ \circledast_{x'\in Loc\setminus\{x\}}([\textsc{Wr}(x')]_r * [\textsc{Rd}(x')]_r) * [\textsc{Wr}(x)]_r * [\textsc{NoRd}(x)]_r\end{array}\right\}_{\{Hp\}}$$

By Lemma .0.29

$\left\{v'.\ v' = () * [\textsc{Wr}]^1_r * \circledast_{x'\in Loc\setminus\{x\}} [\textsc{Rd}(x')]_r * [\textsc{NoRd}(x)]_r\right\}_{\{\textsc{Hp},\textsc{Rg}(r),\textsc{Rf}(x)\}}$

□

Lemma .0.37 (Make type-respecting assignment)**.**

$$\begin{array}{l}\forall r, x, v, \phi.\ \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r,\phi,x)}^{\textsc{Rf}(x)}, \phi(v) \vdash \\ \quad \left\{[\textsc{Wr}]^1_r * \circledast_{x' \in Loc \setminus \{x\}} [\textsc{Rd}(x')]_r * [\textsc{NoRd}(x)]_r\right\} \\ \quad\quad x := v \\ \quad \left\{v'.\ v' = () * [\textsc{Wr}]^1_r * [\textsc{Rd}]^1_r\right\}\end{array}$$

Proof. The proof follows the same outline as above, except for the last line, before closing $\textsc{Rg}(r), \textsc{Rf}(x)$, by having $\phi(v) * [\textsc{NoRd}(x)]_r$ we can use Lemma .0.32 to obtain $\circledast_{x' \in Loc} [\textsc{Rd}(x')]_r$ to which we can use Lemma .0.30 to obtain $[\textsc{Rd}]^1_r$ □

The $\textbf{LR}_{\textbf{Bin}}$ relation

For a pair $x \triangleq (x_1, x_2)$ we have $x_I \triangleq \pi_1(x)$ and $x_S \triangleq \pi_2(x)$ when x_I and x_S is not defined in the context. Similarly, for a pair $X = (X_1, X_2)$, we have $X_\Pi \triangleq \pi_1(X)$ and $X_\Lambda \triangleq \pi_2(X)$.

$$\begin{array}{rl}\textsc{Heap} & \triangleq \exists h.\ heap(h,\gamma) * \lfloor h \rfloor \\ \textsc{Spec}(h_0, e_0) & \triangleq \exists h, e.\ heap_S(h) * mctx(e,\gamma) * (h_0, e_0) \to^* (h, e) \\ \textsc{Ref}(r,\phi,x) & \triangleq \exists v.\ x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v_S * \mathit{effs}(r,\phi,x,v) \\ \textsc{Reg}(r) & \triangleq locs(r) * tokens(r)\end{array}$$

where

$$\begin{array}{rl}\mathit{effs}(r,\phi,x,v) & \triangleq ([\textsc{Wr}(x)]_r \vee (x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} \_ * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} \_)) * \\ & \quad ([\textsc{Rd}(x)]_r \vee ((v_I, v_S) \in \phi * [\textsc{NoRd}(x)]_r)) \\ locs(r) & \triangleq \exists h.\ rheap_I(h_I, r) * rheap_S(h_S, r) * alloc(h, r) * \\ & \quad \circledast_{(l,v) \in h_I} l \mapsto_I v * \circledast_{(l,v) \in h_S} l \mapsto^\gamma_S v * \\ & \quad \circledast_{\{x \mid x \in (Loc \setminus \mathrm{dom}(h_I)) \times (Loc \setminus \mathrm{dom}(h_S))\}} [\textsc{NoRd}(x)]_r \\ tokens(r) & \triangleq ([\textsc{Wr}]^{\pi_{wr}}_r \vee \circledast_{x \in Loc^2} [\textsc{Wr}(x)]_r) * ([\textsc{Rd}]^{\pi_{rd}}_r \vee \circledast_{x \in Loc^2} [\textsc{Rd}(x)]_r) \\ alloc(h,r) & \triangleq ([\textsc{Al}]^1_r * [\textsc{Al}(h_I, h_S)]^{\frac{1}{2}}_r) \vee [\textsc{Al}((h_I, h_S))]^1_r\end{array}$$

For $M \triangleq \mathcal{RN} \overset{\text{fin}}{\longrightarrow} MonoidName$ list:

$$
\begin{aligned}
[\![\mathbf{1}]\!]^M &\triangleq \lambda x.\, x_I = x_S = () \\
[\![\mathsf{int}]\!]^M &\triangleq \lambda x.\, x_I, x_S \in \mathbb{N} \wedge x_I = x_S \\
[\![\tau_1 \times \tau_2]\!]^M &\triangleq \lambda x.\, \exists y_1, y_2, z_1, z_2.\, x_I = (y_1, y_2) \wedge x_S = (z_1, z_2) \wedge \\
&\qquad \triangleright(y_1, z_1) \in [\![\tau_1]\!]^M \wedge \triangleright(y_2, z_2) \in [\![\tau_2]\!]^M \\
[\![\tau_1 + \tau_2]\!]^M &\triangleq \lambda x.\, (\triangleright \exists (y_I, y_S) \in [\![\tau_1]\!]^M.\, x_I = \mathsf{inj}_1\ y_I \wedge x_S = \mathsf{inj}_1\ y_S) \vee \\
&\qquad (\triangleright \exists (y_I, y_S) \in [\![\tau_2]\!]^M.\, x_I = \mathsf{inj}_2\ y_I \wedge x_S = \mathsf{inj}_2\ y_S) \\
[\![\tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M &\triangleq \lambda x.\, \Box \forall y_I, y_S.\, (\triangleright(y_I, y_S) \in [\![\tau_1]\!]^M) \Rightarrow \\
&\qquad \mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(x_I\ y_I, x_S\ y_S) \\
[\![\mathbf{ref}_\rho\ \tau]\!]^M &\triangleq \lambda x.\, \boxed{\textsc{Ref}(M(\rho), [\![\tau]\!]^M, x_I, x_S)}^{\textsc{Rf}(x_I, x_S)} * \\
&\qquad \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))}
\end{aligned}
$$

$$
\begin{aligned}
P_{toks}(\rho, r, \pi, \varepsilon) &\triangleq (\rho \notin \mathsf{rds}\ \varepsilon \vee [\textsc{Rd}]^\pi_r) * (\rho \notin \mathsf{wrs}\ \varepsilon \vee [\textsc{Wr}]^\pi_r) * \\
&\qquad (\rho \notin \mathsf{als}\ \varepsilon \vee [\textsc{Al}]^\pi_r) \\
P_{reg}(R, g, \varepsilon, M) &\triangleq \circledast_{\rho \in R} P_{toks}(\rho, M(\rho), g(\rho), \varepsilon) * \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}(\phi)(e_I, e_S) \triangleq \forall g \in \Pi \to \mathit{Perm}, j : \mathcal{A}, e_0 : \textsc{Exp}, \textsc{Hp}, \textsc{Sp}, h_0. \\
&\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0)}^{\textsc{Sp}} \vdash \\
&\quad \{j \Rightarrow_S e_S * P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M) * P_{reg}(\Pi, g, \varepsilon, M)\} \\
&\quad\ e_I \\
&\quad \{v_I.\, \exists v_S.\, j \Rightarrow_S v_S * \phi(v_I, v_S) * P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M) * P_{reg}(\Pi, g, \varepsilon, M)\}_\top
\end{aligned}
$$

Logical relatedness

$$
\begin{aligned}
&\Pi \mid \Lambda \mid \overline{x : \tau} \models_{\mathrm{Bin}} e_1 \le_{log} e_2 : \tau, \varepsilon \triangleq \\
&\quad \vdash_{\mathrm{Iris}} \forall M. \forall \overline{x_I}, \overline{x_S}.\, \overline{[\![\tau]\!]^M}(\overline{x_I, x_S}) \\
&\qquad \Longrightarrow \mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}([\![\tau]\!]^M)(e_1[\overline{x_I/x}], e_2[\overline{x_S/x}])
\end{aligned}
$$

Theorem .0.38 (Fundamental Theorem)**.** *If* $\Pi \mid \Delta \mid \Gamma \vdash e : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \models_{\mathrm{Bin}} e \le_{log} e : \tau, \varepsilon$

Proof. Proof omitted. □

Theorem .0.39 (Soundness)**.** *If* $\Pi \mid \Delta \mid \Gamma \models_{\mathrm{Bin}} e_I \le_{log} e_S : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_I \le_{ctx} e_S : \tau, \varepsilon$.

Proof. Proof omitted. □

Example: Type violating assignments

Consider the following two programs:

$$e_1 \triangleq (x := ();x := \mathsf{true}) \qquad e_2 \triangleq x := \mathsf{true}$$

We would like to show the following:

$$\cdot \mid \rho \mid x : \mathsf{ref}_\rho\ \mathsf{B} \models_{\text{Bin}} e_1 \preceq e_2 : \mathbf{1}, \left\{wr_\rho, rd_\rho\right\}$$

which means that we have to show:

$$\mathcal{E}^{\Pi;\Lambda}_{\{wr_\rho, rd_\rho\},M}(\llbracket \mathbf{1} \rrbracket^M)(e_1, e_2)$$

Lemma .0.40.

$$\forall r.\ \textsc{Reg}(r) * [\textsc{Rd}]^1_r \Longleftrightarrow \textsc{Reg}(r) * \circledast_{x \in \text{Loc}^2} [\textsc{Rd}(x)]_r$$

Lemma .0.41.

$$\begin{array}{l}\forall r, \pi, \phi, x, v.\\ \quad \{[\textsc{Wr}]^\pi_r * [\textsc{Rd}(x)]_r * \textsc{Ref}(r, \triangleright\phi, x) * \textsc{Reg}(r) * \textsc{Heap}\}\\ \quad\quad x := v\\ \quad \{w.\ w = () * [\textsc{Wr}]^\pi_r * [\textsc{NoRd}(x)]_r * \textsc{Ref}(r, \phi, x) * \textsc{Reg}(r) * \textsc{Heap}\}\end{array}$$

Proof. Follows from view-shifts shown in the article and appendix □

Lemma .0.42.

$$\begin{array}{l}\forall j, r, \pi, \phi, x, v.\\ \quad \{j \Rightarrow_S x_S := v_S * [\textsc{Wr}]^\pi_r * [\textsc{NoRd}(x)]_r * \textsc{Ref}(r, \triangleright\phi, x) * \textsc{Reg}(r) * \textsc{Heap} * \phi(v_I, v_S)\}\\ \quad\quad x := v_I\\ \quad \{w.\ w = () * j \Rightarrow_S () * [\textsc{Wr}]^\pi_r * [\textsc{Rd}(x)]_r * \textsc{Ref}(r, \phi, x) * \textsc{Reg}(r) * \textsc{Heap}\}\end{array}$$

Proof. Follows from view-shifts shown in the article and appendix □

Context: $x, j, M, \rho, \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}}^{\textsc{Sp}}$
// Let $r = M(\rho)$ and $R = \{\textsc{Hp}, \textsc{Sp}, \textsc{Rf}(x), \textsc{Rg}(r)\}$
$\left\{j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \boxed{\textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}\right\}_R$

Bind on $x_I := (); x_I := \mathsf{true}$

$\left\{j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \boxed{\textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}\right\}_R$

Open R

$\left\{j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \triangleright \textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x) * \triangleright \textsc{Reg}(r) * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}\right\}$

// Follows from VSTimeless

$\left\{j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \textsc{Ref}(r, \triangleright [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec}\right\}$

// Follows from Lemma .0.40

$\left\{j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Wr}]^1_r * \textsc{Ref}(r, \triangleright [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec} * \circledast_{x \in \mathrm{Loc}^2} [\textsc{Rd}(x)]_r\right\}$

$x := ()$

// Follows from Lemma .0.41

$\left\{w.\ w = () * j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Wr}]^1_r * \textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec} * \circledast_{y \in \mathrm{Loc}^2 \setminus \{x\}} [\textsc{Rd}(y)]_r * [\textsc{NoRd}(x)]_r\right\}$

$\left\{w.\ w = () * j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Wr}]^1_r * \boxed{\textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \circledast_{y \in \mathrm{Loc}^2 \setminus \{x\}} [\textsc{Rd}(y)]_r * [\textsc{NoRd}(x)]_r\right\}_R$

Open R

$\left\{j \Rightarrow_S x_S := \mathsf{true} * [\textsc{Wr}]^1_r * \textsc{Ref}(r, \triangleright [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec} * \circledast_{y \in \mathrm{Loc}^2 \setminus \{x\}} [\textsc{Rd}(y)]_r * [\textsc{NoRd}(x)]_r\right\}$

$x := \mathsf{true}$

// Follows from Lemma .0.42

$\left\{w'.\ w' = () * j \Rightarrow_S () * [\textsc{Wr}]^1_r * \textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec} * \circledast_{y \in \mathrm{Loc}^2 \setminus \{x\}} [\textsc{Rd}(y)]_r * [\textsc{Rd}(x)]_r\right\}$

$\left\{w'.\ w' = () * j \Rightarrow_S () * [\textsc{Wr}]^1_r * \textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec} * \circledast_{y \in \mathrm{Loc}^2} [\textsc{Rd}(y)]_r\right\}$

// Follows from Lemma .0.40

$\left\{w'.\ w' = () * j \Rightarrow_S () * [\textsc{Wr}]^1_r * \textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x) * \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec} * [\textsc{Rd}]^1_r\right\}$

$\left\{w'.\ w' = () * j \Rightarrow_S () * [\textsc{Wr}]^1_r * \boxed{\textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Rd}]^1_r\right\}_R$

$\left\{w'.\ \exists w_S.\ j \Rightarrow_S w_S * [\textsc{Wr}]^1_r * \boxed{\textsc{Ref}(r, [\![\mathbf{1}]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Rd}]^1_r * [\![\mathbf{1}]\!]^M(w', w_S)\right\}_R$

Example: Local state

We have intensionally defined our logical relations to support local state that is not tracked by the type-and-effect system. This means that we can for instance prove that a pure expression approximates an impure expression at a pure effect type, because the impure expression uses untracked local state. To illustrate, consider the following two functions:

$$e_1 \triangleq \mathsf{true} \qquad e_2 \triangleq \mathsf{let}\ x = \mathsf{new}\ \mathsf{true}\ \mathsf{in}\ !x$$

thus we would like to show:

$$\cdot \mid \cdot \mid \cdot \vDash_{\text{Eff}} e_1 \preceq e_2 : \mathsf{B}, \emptyset$$

Context: $\boxed{\textsc{Heap}}^{\text{Hp}}, \boxed{\textsc{Spec}}^{\text{Sp}}$

$\{j \Rightarrow_S e_2\}$

Open Sp:

- $\{\textsc{Spec} * j \Rightarrow_S e_2\}$
- $\left\{\textsc{Spec} * \exists v_S.\ j \Rightarrow_S\ !v_S * v_S \mapsto_S \mathsf{true}\right\}$
- $\left\{\textsc{Spec} * \exists v_S, v_S'.\ j \Rightarrow_S v_S' * v_S \mapsto_S \mathsf{true} * v_S' = \mathsf{true}\right\}$

$\left\{\exists v_S, v_S'.\ j \Rightarrow_S v_S' * v_S' = \mathsf{true}\right\}$

true

$\left\{v_I.\ v_I = \mathsf{true} * \exists v_S, v_S'.\ j \Rightarrow_S v_S' * v_S' = \mathsf{true}\right\}$

$\left\{v_I.\ \exists v_S'.\ j \Rightarrow_S v_S' * (v_I, v_S') \in [\![\mathsf{B}]\!]^M\right\}$

As a consequence of this choice to allow local state not tracked by the type-and-effect system, it is possible to have non-determinism in expressions that we deem semantically pure. For instance, the following expression returns 1 or 2 non-deterministically, but can be proven to be semantically pure, because it only uses local state.

$$e \triangleq \mathsf{let}\ x = \mathsf{new}\ 0\ \mathsf{in}\ x := 1 \,||\, x := 2; !x$$

The LR_{PAR} relation

For a pair $x \triangleq (x_1, x_2)$ we have $x_I \triangleq \pi_1(x)$ and $x_S \triangleq \pi_2(x)$ when x_I and x_S is not defined in the context. Similarly, for a pair $X = (X_1, X_2)$, we have $X_\Pi \triangleq \pi_1(X)$ and $X_\Lambda \triangleq \pi_2(X)$. We assume a list of monoid-names γ to be defined globally.

A spec can either be active ($\pi < 1$) or finished ($\pi = 1$).

$$
\begin{aligned}
\textsc{Heap} &\triangleq \exists h_I.\ heap_I(h_I) * \lfloor h_I \rfloor \\
\textsc{Ref}(r,\phi,x) &\triangleq \exists v.\ ref(r,\phi,x,v) \\
\textsc{Reg}(r) &\triangleq \exists h.\ locs(h,r) * toks(1,1,r) \\
\textsc{Spec}(h_0,e_0,\zeta) &\triangleq \exists h,e.\, heap_S(h,\zeta) * mctx(e,\zeta) * (h_0,e_0) \rightarrow^* (h,e) * \\
&\qquad ([\textsc{Sr}]^1_\zeta \vee ([\textsc{Sr}]^{\frac{1}{2}}_\zeta * disj_H(h_0,h)))
\end{aligned}
$$

where

$$
\begin{aligned}
ref(r,\phi,x,v) &\triangleq x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v_S * effs(r,\phi,x,v) \\
effs(r,\phi,x,v) &\triangleq ([\textsc{Wr}(x)]_r \vee (x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} \_ * x_S \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} \_)) * \\
&\qquad ([\textsc{Rd}(x)]_r \vee (\phi(v_I,v_S) * [\textsc{NoRd}(x)]_r)) \\
locs(h,r) &\triangleq \exists S.\ locs(h,r,S,S) \\
locs(h,r,S,S') &\triangleq rheap_I(h_I,r) * rheap_S(h_S,r) * alloc(h,r) * \\
&\qquad slink(r,S,h_S,\tfrac{1}{2},\tfrac{1}{4}) * \circledast_{(l,v)\in h_I} l \mapsto v * \\
&\qquad \circledast_{\zeta\in S'} \circledast_{(l,v)\in h_S} l \mapsto^{\zeta}_{S} v * \\
&\qquad \circledast_{x\in(Loc\setminus \mathrm{dom}(h_I))\times(Loc\setminus\mathrm{dom}(h_S))} [\textsc{NoRd}(x)]_r \\
slink(r,S,h,\pi,\pi') &\triangleq ([\textsc{Mu}(r,S)]^{\pi} \vee [\textsc{Im}(r,S,h)]^{\pi'}) \\
toks(\pi_{rd},\pi_{wr},r) &\triangleq ([\textsc{Wr}]^{\pi_{wr}}_r \vee \circledast_{x\in Loc^2} [\textsc{Wr}(x)]_r) * \\
&\qquad ([\textsc{Rd}]^{\pi_{rd}}_r \vee \circledast_{x\in Loc^2} [\textsc{Rd}(x)]_r) \\
alloc(h,r) &\triangleq ([\textsc{Al}]^1_r * [\textsc{Al}(h_I,h_S)]^{\frac{1}{2}}_r) \vee [\textsc{Al}(h_I,h_S)]^1_r \\
disj_H(h_0,h) &\triangleq \exists h_Y.\ [h_Y]_H \wedge \mathrm{dom}(h_0) \cap h_Y = \emptyset \wedge \\
&\qquad (\mathrm{dom}(h) \setminus \mathrm{dom}(h_0)) \subset h_Y
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathbf{1}]\!]^M &\triangleq \lambda x.\ x_I = x_S = () \\
[\![\mathbf{B}]\!]^M &\triangleq \lambda x.\ x_I, x_S \in \{\mathbf{true},\mathbf{false}\} \wedge x_I = x_S \\
[\![\mathbf{int}]\!]^M &\triangleq \lambda x.\ x_I, x_S \in \mathbb{N} \wedge x_I = x_S \\
[\![\tau_1 \times \tau_2]\!]^M &\triangleq \lambda x.\ \exists y_1,y_2,z_1,z_2.\ x_I = (y_1,y_2) \wedge x_S = (z_1,z_2) \wedge \\
&\qquad \triangleright(y_1,z_1) \in [\![\tau_1]\!]^M \wedge \triangleright(y_2,z_2) \in [\![\tau_2]\!]^M \\
[\![\tau_1 + \tau_2]\!]^M &\triangleq \lambda x.\ (\triangleright \exists (y_I,y_S) \in [\![\tau_1]\!]^M.\ x_I = \mathbf{inj}_1\ y_I \wedge x_S = \mathbf{inj}_1\ y_S) \vee \\
&\qquad (\triangleright \exists (y_I,y_S) \in [\![\tau_2]\!]^M.\ x_I = \mathbf{inj}_2\ y_I \wedge x_S = \mathbf{inj}_2\ y_S) \\
[\![\tau_1 \rightarrow^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M &\triangleq \lambda x.\ \Box \forall y_I, y_S.\ (\triangleright(y_I,y_S) \in [\![\tau_1]\!]^M) \Rightarrow \\
&\qquad \mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(x_I\ y_I, x_S\ y_S) \\
[\![\mathbf{ref}_\rho\ \tau]\!]^M &\triangleq \lambda x.\ \boxed{\textsc{Ref}(M(\rho),[\![\tau]\!]^M,x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))}
\end{aligned}
$$

$$
\begin{aligned}
P_{par}(R,g,\varepsilon,M,\zeta) &\triangleq \circledast_{\rho\in mutable(R,g,\varepsilon)} [\textsc{Mu}(M(\rho),\{\zeta\})]^{g(\rho)} * \\
&\qquad \circledast_{\rho\in R\setminus mutable(R,g,\varepsilon)} \exists S.\, slink(M(\rho),\{\zeta\}\uplus S,h,g(\rho),g(\rho)) \\
P_{toks}(\rho,r,\pi,\varepsilon) &\triangleq (\rho\notin \mathsf{rds}\ \varepsilon \vee [\textsc{Rd}]^{\pi}_{r}) * (\rho\notin \mathsf{wrs}\ \varepsilon \vee [\textsc{Wr}]^{\pi}_{r}) * \\
&\qquad (\rho\notin \mathsf{als}\ \varepsilon \vee [\textsc{Al}]^{\pi}_{r}) \\
P_{reg}(R,g,\varepsilon,M,\zeta) &\triangleq P_{par}(R,\tfrac{\mathbf{1}}{2}\circ g,\varepsilon,M,\zeta) * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\
&\qquad \circledast_{\rho\in R} P_{toks}(\rho,M(\rho),g(\rho),\varepsilon) \\
mutable(R,g,\varepsilon) &\triangleq \mathsf{wrs}\ \varepsilon \cup \mathsf{als}\ \varepsilon \cup \left\{\rho \mid \rho\in R \wedge g(\rho)=\tfrac{1}{2}\right\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}(\phi)(e_I,e_S) \triangleq \forall g\in\Pi\to Perm, j\in\mathcal{A}, e_0\in \textsc{Exp}, h_0,\pi,\zeta. \\
&\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(e_0,h_0,\zeta)}^{\textsc{Sp}(\zeta)} \vdash \\
&\left\{ j \overset{\zeta}{\Rightarrow}_S e_S * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda,\mathbf{1},\varepsilon,M,\zeta) * P_{reg}(\Pi,g,\varepsilon,M,\zeta) \right\} \\
&\quad e_I \\
&\left\{ v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda,\mathbf{1},\varepsilon,M,\zeta) * P_{reg}(\Pi,g,\varepsilon,M,\zeta) * \phi(v_I,v_S) \right\}_{\top}
\end{aligned}
$$

Logical relatedness

$$
\begin{aligned}
&\Pi \mid \Lambda \mid \overline{x:\tau} \models_{\textsc{Par}} e_1 \le_{log} e_2 : \tau,\varepsilon \triangleq \\
&\quad \vdash_{\textsc{Iris}} \forall M.\forall \overline{x_I},\overline{x_S}.\, \overline{[\![\tau]\!]^M}(\overline{x_I,x_S}) \\
&\qquad \Longrightarrow \mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}([\![\tau]\!]^M)(e_1[\overline{x_I/x}], e_2[\overline{x_S/x}])
\end{aligned}
$$

Theorem .0.43 (Soundness). *If* $\Pi \mid \Delta \mid \Gamma \models_{\textsc{Bin}} e_I \le_{log} e_S : \tau,\varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_I \le_{ctx} e_S : \tau,\varepsilon$.

Proof. Proof in end of appendix. □

Fundamental Theorem

Theorem .0.44 (Fundamental Theorem)**.** *If* $\Pi \mid \Delta \mid \Gamma \vdash e : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \models_{\text{BIN}} e \leq_{log} e : \tau, \varepsilon$

Proof. Hard cases are shown below □

We will use the predicates below to make proving specific properties about their internal state easier. The intended meaning and naming remains.

$$\begin{aligned}\textsc{Spec}(h_0, h, e_0, e, \pi, \zeta) \triangleq\ & heap_S(h, \zeta) * mctx(e, \zeta) * (h_0, e_0) \rightarrow^* (h, e) * [\textsc{Sr}]^{\pi}_{\zeta} * \\ & (\pi = 1 \vee (\pi < 1 * disj_H(h_0, h)))\end{aligned}$$

$$\textsc{Spec}(h_0, e_0, \zeta) \triangleq \exists h, e.\ \textsc{Spec}(h_0, h, e_0, e, \frac{1}{2}, \zeta)$$

$$\begin{aligned}S(\zeta, j, h_0, e_0, e, \pi, R, g, \varepsilon, M) \triangleq\ & \boxed{\textsc{Spec}(e_0, h_0, \zeta)}^{\text{SP}(\zeta)} * j \overset{\zeta}{\Rightarrow}_S e * [\textsc{Sr}]^{\pi}_{\zeta} * \\ & P_{reg}(R_\Lambda, \mathbf{1}, \varepsilon, M, \zeta) * P_{reg}(R_\Pi, g, \varepsilon, M, \zeta)\end{aligned}$$

Open invariants

Lemma .0.45 (Can remove ▷)**.**

$$\triangleright \textsc{Heap} \Rrightarrow \textsc{Heap} \tag{1}$$

$$\forall \zeta.\ \triangleright \textsc{Spec}(h_0, e_0, \zeta) \Rrightarrow \textsc{Spec}(h_0, e_0, \zeta) \tag{2}$$

$$\forall r.\ \triangleright \textsc{Reg}(r) \Rrightarrow \textsc{Reg}(r) \tag{3}$$

$$\forall r, \phi, x.\ \triangleright \textsc{Ref}(r, \phi, x) \Rrightarrow \textsc{Ref}(r, \triangleright \phi, x) \tag{4}$$

Proof. ▷ commute over $*$ and all assertions inside are either ghost-resource or pure statements thus we can use Timeless to remove the ▷. □

Specification reduction

Lemma .0.46 (Specification reduction / no allocation)**.**

$$\begin{aligned}&\forall j, e_0, e, e_1, e_1', e_1', \pi, \pi', h_0, h, h', K, \zeta.\\ &\quad (heap_S(h, \zeta) * disj_H(h_0, h) \Rrightarrow heap_S(h', \zeta) * disj_H(h_0, h') \Rightarrow \\ &\quad \textsc{Spec}(h_0, h, e_0, e, \pi, \zeta) * [\textsc{Sr}]^{\pi'}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S K[e_1] * (h, e_1) \rightarrow (h', e_1')\\ \Rightarrow\quad &\exists e'.\ \textsc{Spec}(h_0, h', e_0, e, \pi, \zeta) * [\textsc{Sr}]^{\pi'}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S K[e_1']\end{aligned}$$

Proof.

$$
\begin{aligned}
& \textsc{Spec}(h_0, h, e_0, e, \pi, \zeta) * [\textsc{Sr}]^{\pi'}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S K[e_1] * \\
& (h, e_1) \to (h', e_1') \\
(\text{unfold}) \Rrightarrow \quad & heap_S(h, \zeta) * mctx(e, \zeta) * (h_0, e_0) \to^* (h, e) * [\textsc{Sr}]^{\pi}_{\zeta} * \\
& (\pi = 1 \vee (\pi < 1 * disj_H(h_0, h))) * [\textsc{Sr}]^{\pi'}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S K[e_1] * \\
& (h, e_1) \to (h', e_1') \\
\Rrightarrow \quad & heap_S(h, \zeta) * mctx(e, \zeta) * (h_0, e_0) \to^* (h, e) * [\textsc{Sr}]^{\pi+\pi'}_{\zeta} * \\
& disj_H(h_0, h) * j \overset{\zeta}{\Rightarrow}_S K[e_1] * (h, e_1) \to (h', e_1') \\
(\textit{Lemma } .0.21) \Rrightarrow \quad & \exists k.\ heap_S(h, \zeta) * mctx(e, \zeta) * (h_0, e_0) \to^* (h, e) * \\
& [\textsc{Sr}]^{\pi+\pi'}_{\zeta} * disj_H(h_0, h) * j \overset{\zeta}{\Rightarrow}_S K[k] * \\
& (h, e_1) \to (h', e_1') * k \overset{\zeta}{\Rightarrow}_S e_1 \\
(\textit{Lemma } .0.24) \Rrightarrow \quad & \exists k, e'.\ heap_S(h, \zeta) * mctx(e', \zeta) * (h_0, e_0) \to^* (h', e') * \\
& [\textsc{Sr}]^{\pi+\pi'}_{\zeta} * disj_H(h_0, h) * j \overset{\zeta}{\Rightarrow}_S K[k] * \\
& (h, e_1) \to (h', e_1') * k \overset{\zeta}{\Rightarrow}_S e_1' \\
(ass) \Rrightarrow \quad & \exists k, e'.\ heap_S(h', \zeta) * mctx(e', \zeta) * (h_0, e_0) \to^* (h', e') * \\
& [\textsc{Sr}]^{\pi+\pi'}_{\zeta} * disj_H(h_0, h') * j \overset{\zeta}{\Rightarrow}_S K[k] * \\
& (h, e_1) \to (h', e_1') * k \overset{\zeta}{\Rightarrow}_S e_1' \\
(\textit{Lemma } .0.22) \Rrightarrow \quad & \exists k, e'.\ heap_S(h', \zeta) * mctx(e', \zeta) * (h_0, e_0) \to^* (h', e') * \\
& [\textsc{Sr}]^{\pi+\pi'}_{\zeta} * disj_H(h_0, h') * j \overset{\zeta}{\Rightarrow}_S K[e_1'] * \\
& (h, e_1) \to (h', e_1') \\
(\text{fold}) \Rrightarrow \quad & \exists e'.\ \textsc{Spec}(h_0, h', e_0, e', \pi, \zeta) * [\textsc{Sr}]^{\pi'}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S K[e_1']
\end{aligned}
$$

□

Lemma .0.47 (Spec pure reduction step)**.**

$$
\begin{aligned}
& \forall e_1, e_1', h, K, \pi.\ (h, e_1) \to (h, e_1') \Rightarrow \\
& \quad \forall \zeta, j.\ (\boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)} * j \overset{\zeta}{\Rightarrow}_S K[e_1] * [\textsc{Sr}]^{\pi}_{\zeta} \Rrightarrow_{\textsc{Sp}(\zeta)} \\
& \qquad \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)} * j \overset{\zeta}{\Rightarrow}_S K[e_1'] * [\textsc{Sr}]^{\pi}_{\zeta})
\end{aligned}
$$

Proof.

$$\begin{array}{rl}
& \boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)} * j \overset{\zeta}{\Rightarrow}_S K[e_1] * [\textsc{Sr}]^{\pi}_{\zeta} \\
(\text{VSInv})\ {}^{\textsc{Sp}(\zeta)}\!\Rightarrow^{\emptyset} & \triangleright\, \textsc{Spec}(h_0,e_0,\zeta) * j \overset{\zeta}{\Rightarrow}_S K[e_1] * [\textsc{Sr}]^{\pi}_{\zeta} \\
(\textit{Lemma .0.45})\Rightarrow & \textsc{Spec}(h_0,e_0,\zeta) * j \overset{\zeta}{\Rightarrow}_S K[e_1] * [\textsc{Sr}]^{\pi}_{\zeta} \\
(\text{unfold})\Rightarrow & \exists h, e.\ heap_S(h,\zeta) * mctx(e,\zeta) * (h_0,e_0) \rightarrow^* (h,e) * \\
& ([\textsc{Sr}]^{1}_{\zeta} \vee ([\textsc{Sr}]^{\frac{1}{2}}_{\zeta} * disj_H(h_0,h))) * \\
& j \overset{\zeta}{\Rightarrow}_S K[e_1] * [\textsc{Sr}]^{\pi}_{\zeta} \\
(\textit{Lemma .0.46})\Rightarrow & \exists h, e'.\ heap_S(h,\zeta) * mctx(e',\zeta) * (h_0,e_0) \rightarrow^* (h,e') * \\
& ([\textsc{Sr}]^{1}_{\zeta} \vee ([\textsc{Sr}]^{\frac{1}{2}}_{\zeta} * disj_H(h_0,h))) * \\
& j \overset{\zeta}{\Rightarrow}_S K[e_1'] * [\textsc{Sr}]^{\pi}_{\zeta} \\
(\text{fold})\Rightarrow & \textsc{Spec}(h_0,e_0,\zeta) * j \overset{\zeta}{\Rightarrow}_S K[e_1'] * [\textsc{Sr}]^{\pi}_{\zeta} \\
(\text{VSClose}) & \boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)} * j \overset{\zeta}{\Rightarrow}_S K[e_1'] * [\textsc{Sr}]^{\pi}_{\zeta}
\end{array}$$

□

Function abstraction

Lemma .0.48. *If*

$$[\![\tau_1 \rightarrow^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M(f_I,f_S) \vdash \mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}(\tau_2)(e_I,e_S) \quad (H1)$$

then

$$[\![\tau_1 \rightarrow^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M(\textbf{rec}\ f(x).e_I, \textbf{rec}\ f(x).e_S)$$

Proof. Löb-induction, thus we have to show:

$$\square \forall y_I, y_S.\ (\triangleright(y_I,y_S) \in [\![\tau_1]\!]^M) \Rightarrow \mathcal{E}^{\Pi;\Lambda}_{\varepsilon,M}([\![\tau_2]\!]^M)(\textbf{rec}\ f(x).e_I\ y_I, \textbf{rec}\ f(x).e_S\ y_S)$$

under the assumption $\triangleright([\![\tau_1 \rightarrow^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M(\textbf{rec}\ f(x).e_I, \textbf{rec}\ f(x).e_S))$:

Context: $h_0, e_0, j, \zeta, \pi, g, \triangleright([\![\tau_1]\!]^M(y_I,y_S)), \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)}$

Context: $\triangleright([\![\tau_1 \rightarrow^{\Pi,\Lambda}_{\varepsilon} \tau_2]\!]^M(f_I,f_S))$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S \textbf{rec}\ f(x).e_S\ y_S * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M, \zeta) * \\ P_{reg}(\Pi, g, \varepsilon, M, \zeta) \end{array}\right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta)\}}$$

$\textbf{rec}\ f(x).e_I\ y_I$

$$\left\{\begin{array}{l} v_I.\ \exists v_S.\ [\![\tau_2]\!]^M(v_I,v_S) * j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi}_{\zeta} * \\ P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M, \zeta) * P_{reg}(\Pi, g, \varepsilon, M, \zeta) \end{array}\right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta)\}}$$

We can take a step, thereby remove the ▷ from the context

$$\text{Context: } h_0, e_0, j, \zeta, \pi, g, \triangleright(\llbracket \tau_1 \rrbracket^M(y_I, y_S)), \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}$$
$$\text{Context: } (\llbracket \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2 \rrbracket^M(f_I, f_S))$$
$$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_S[y_S/x, f_S/f] * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M, \zeta) * \\ P_{reg}(\Pi, g, \varepsilon, M, \zeta) \end{array} \right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta)\}}$$
$$e_I[y_I/x, f_I/f]$$
$$\left\{ \begin{array}{l} v_I.\ \exists v_S.\ \llbracket \tau_2 \rrbracket^M(v_I, v_S) * j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \mathbf{1}, \varepsilon, M, \zeta) * \\ P_{reg}(\Pi, g, \varepsilon, M, \zeta) \end{array} \right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta)\}}$$

Now we can apply $H1$ with y_I and y_S.

□

Function application

Lemma .0.49.

$$\forall v_1, v_2, j, \pi, \Lambda, \Pi, \varepsilon, h_0, e_0, \zeta, M.$$
$$\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \llbracket \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2 \rrbracket^M(v_{1I}, v_{1S}), \llbracket \tau_1 \rrbracket^M(v_{2I}, v_{2S}) \vdash$$
$$\left\{ j \overset{\zeta}{\Rightarrow}_S v_{1S}\ v_{2S} * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \boldsymbol{1}, \varepsilon, M, \zeta) * P_{reg}(\Pi, g, \varepsilon, M, \zeta) \right\}$$
$$v_{1I}\ v_{2I}$$
$$\left\{ \begin{array}{l} v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S v_S * \llbracket \tau_2 \rrbracket^M(v_I, v_S) * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \boldsymbol{1}, \varepsilon, M, \zeta) * \\ \quad P_{reg}(\Pi, g, \varepsilon, M, \zeta) \end{array} \right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta)\}}$$

Proof. Unfolding $\llbracket \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2 \rrbracket^M(v_{1I}, v_{1S})$ and apply that the computations are related, thus we have to show $\llbracket \tau_1 \rrbracket^M(v_{2I}, v_{2S})$, which we have from our assumption. □

Par

$$\mathsf{regs}(\varepsilon) \triangleq \{r \mid r \in \mathsf{rds}\ \varepsilon \cup \mathsf{wrs}\ \varepsilon \cup \mathsf{als}\ \varepsilon\}$$

Lemma .0.50 (Splitting region)**.**

$$\begin{array}{ll} & \forall R_1, R_2, g, \varepsilon_1, \varepsilon_2, M, \zeta. \\ & \quad P_{reg}(R_1 \uplus R_2, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * \mathsf{regs}(\varepsilon_1) \subseteq R_1 * \mathsf{regs}(\varepsilon_2) \subseteq R_2 \\ \Rrightarrow & \quad P_{reg}(R_1, g, \varepsilon_1, M, \zeta) * P_{reg}(R_2, g, \varepsilon_2, M, \zeta) \end{array}$$

Lemma .0.51 (Assembling regions)**.**

$$\begin{array}{ll} & \forall R_1, R_2, g, \varepsilon_1, \varepsilon_2, M, \zeta. \\ & \quad P_{reg}(R_1, g, \varepsilon_1, M, \zeta) * P_{reg}(R_2, g, \varepsilon_2, M, \zeta) \\ \Rrightarrow & \quad P_{reg}(R_1 \uplus R_2, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}$$

Lemma .0.52 (Changing region)**.**

$$\begin{array}{ll} & \forall R, g, \varepsilon_1, \varepsilon_2, M, \zeta. \\ & \quad P_{reg}(R, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \\ \iff & \quad P_{reg}(R, \frac{g}{2}, \varepsilon_1, M, \zeta) * P_{reg}(R, \frac{g}{2}, \varepsilon_2, M, \zeta) * P_{reg}(R, \frac{g}{2}, \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) \end{array}$$

where

$$\tfrac{g}{2}(\rho) \triangleq \begin{cases} \frac{g(\rho)}{2} & \rho \in \mathrm{dom}(g) \\ \bot & \textit{otherwise} \end{cases}$$

Lemma .0.53 (New expressions in evaluation contexts)**.**

$$\forall j, e_1, e_2.\ j \overset{\zeta}{\Rightarrow}_S e_1 \,\|\, e_2 \Rrightarrow \exists k_1, k_2.\ j \overset{\zeta}{\Rightarrow}_S k_1 \,\|\, k_2 * k_1 \overset{\zeta}{\Rightarrow}_S e_1 * k_2 \overset{\zeta}{\Rightarrow}_S e_2$$

Proof. Follows from Lemma .0.21. □

Lemma .0.54 (Substituting expressions in evaluation contexts)**.**

$$\forall j, k_1, j_2, v_1, v_2.\ j \overset{\zeta}{\Rightarrow}_S k_1 \,\|\, k_2 * k_1 \overset{\zeta}{\Rightarrow}_S v_1 * k_2 \overset{\zeta}{\Rightarrow}_S v_2 \Rrightarrow j \overset{\zeta}{\Rightarrow}_S v_1 \,\|\, v_2$$

Proof. Follows from Lemma .0.22. □

Lemma .0.55 (Par)**.**

$$\begin{array}{l} \forall j, h_0, e_0, e_1, e_2, \zeta, \pi, \Lambda_1, \Lambda_2, \Lambda_3, \Pi, \varepsilon_1, \varepsilon_2, M, g, \tau_1, \tau_2. \\ \quad \mathit{regs}(\varepsilon_1) \subseteq \Lambda_1 \cup \Lambda_3 \cup \Pi \wedge \mathit{regs}(\varepsilon_2) \subseteq \Lambda_2 \cup \Lambda_3 \cup \Pi \Rightarrow \\ \quad (\mathcal{E}^{(\Pi,\Lambda_3);\Lambda_1}_{\varepsilon_1,M}([\![\tau_1]\!]^M)(e_{1I}, e_{1S}), \mathcal{E}^{(\Pi,\Lambda_3);\Lambda_2}_{\varepsilon_2,M}([\![\tau_2]\!]^M)(e_{2I}, e_{2S}), \\ \quad \boxed{\textsc{Heap}}^{\mathrm{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\mathrm{Sp}(\zeta)} \vdash \\ \qquad \mathcal{E}^{\Pi,(\Lambda_1,\Lambda_2,\Lambda_3)}_{\varepsilon_1 \cup \varepsilon_2, M}([\![\tau_1 \times \tau_2]\!]^M)(e_{1I} \,\|\, e_{2I}, e_{1S} \,\|\, e_{2S})) \end{array}$$

Proof. Let $R = \{\textsc{Hp}, \textsc{Sp}(\zeta)\}$.

Context: $j, h_0, e_0, e_1, e_2, \zeta, \pi, \Lambda_1, \Lambda_2, \Lambda_3, \Pi, \varepsilon_1, \varepsilon_2, M, g, \tau_1, \tau_2$

Context: $\mathcal{E}^{(\Pi,\Lambda_3);\Lambda_1}_{\varepsilon_1,M}(\llbracket\tau_1\rrbracket^M)(e_{1I}, e_{1S}), \mathcal{E}^{(\Pi,\Lambda_3);\Lambda_2}_{\varepsilon_2,M}(\llbracket\tau_2\rrbracket^M)(e_{2I}, e_{2S})$

Context: $\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} \,\|\, e_{2S} * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * \\ P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}$$

// Lemma .0.50

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} \,\|\, e_{2S} * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// Lemma .0.52

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} \,\|\, e_{2S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3, \frac{\mathbf{1}}{2}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_3, \frac{\mathbf{1}}{2}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3, \frac{\mathbf{1}}{2}, \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) * \\ P_{reg}(\Pi, \frac{g}{2}, \varepsilon_1, M, \zeta) * P_{reg}(\Pi, \frac{g}{2}, \varepsilon_2, M, \zeta) * P_{reg}(\Pi, \frac{g}{2}, \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// Let $g'(r) \triangleq \begin{cases} \frac{g}{2} & r \in \Pi \\ \frac{\mathbf{1}}{2} & r \in \Lambda_3 \\ \bot & \texttt{otherwise} \end{cases}$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} \,\|\, e_{2S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// Lemma .0.53

$$\left\{\begin{array}{l} \exists k_1, k_2.\ j \overset{\zeta}{\Rightarrow}_S k_1 \,\|\, k_2 * k_1 \overset{\zeta}{\Rightarrow}_S e_{1S} * k_2 \overset{\zeta}{\Rightarrow}_S e_{2S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * \\ P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

Frame

$$\left\{\begin{array}{l} \exists k_1, k_2.\ k_1 \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) * k_2 \overset{\zeta}{\Rightarrow}_S e_{2S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

$e_{1I} \| e_{2I}$

$$\left\{\begin{array}{l} \exists k_1.\ k_1 \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) \end{array}\right\}_R$$

e_{1I}

$$\left\{\begin{array}{l} v_{1I}.\ \exists k_1, v_{1S}.\ k_1 \overset{\zeta}{\Rightarrow}_S v_{1S} * \llbracket\tau_1\rrbracket^M(v_{1I}, v_{1S}) * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * \\ P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} \exists k_2.\ k_2 \overset{\zeta}{\Rightarrow}_S e_{2S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

e_{2I}

$$\left\{\begin{array}{l} v_{2I}.\ \exists k_2, v_{2S}.\ k_2 \overset{\zeta}{\Rightarrow}_S v_{2S} * \llbracket\tau_2\rrbracket^M(v_{2I}, v_{2S}) * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} v_I.\ \exists k_1, k_2, v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * \llbracket\tau_1\rrbracket^M(v_{1I}, v_{1S}) * \llbracket\tau_2\rrbracket^M(v_{2I}, v_{2S}) * \\ k_1 \overset{\zeta}{\Rightarrow}_S v_{1S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) * \\ k_2 \overset{\zeta}{\Rightarrow}_S v_{2S} * [\textsc{Sr}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} v_I.\ \exists k_1, k_2, v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * \\ j \overset{\zeta}{\Rightarrow}_S v_{1S} \,\|\, v_{2S} * k_1 \overset{\zeta}{\Rightarrow}_S v_{1S} * k_2 \overset{\zeta}{\Rightarrow}_S v_{2S} * [\mathrm{SR}]^{\frac{\pi}{2}}_{\zeta} * [\mathrm{SR}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// Lemma .0.54

$$\left\{\begin{array}{l} v_I.\ \exists v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * \\ j \overset{\zeta}{\Rightarrow}_S v_{1S} \,\|\, v_{2S} * [\mathrm{SR}]^{\frac{\pi}{2}}_{\zeta} * [\mathrm{SR}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3 \uplus \Pi, g', \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// Lemma .0.52

$$\left\{\begin{array}{l} v_I.\ \exists v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * \\ j \overset{\zeta}{\Rightarrow}_S v_{1S} \,\|\, v_{2S} * [\mathrm{SR}]^{\frac{\pi}{2}}_{\zeta} * [\mathrm{SR}]^{\frac{\pi}{2}}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// From $\mathrm{regs}(\varepsilon_1) \notin \Lambda_2$ and $\mathrm{regs}(\varepsilon_2) \notin \Lambda_1$

$$\left\{\begin{array}{l} v_I.\ \exists v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * \\ j \overset{\zeta}{\Rightarrow}_S v_{1S} \,\|\, v_{2S} * [\mathrm{SR}]^{\pi}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2 \cup \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} v_I.\ \exists v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * \\ j \overset{\zeta}{\Rightarrow}_S v_{1S} \,\|\, v_{2S} * [\mathrm{SR}]^{\pi}_{\zeta} * P_{reg}(\Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * \\ P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

// Pure step

$$\left\{\begin{array}{l} v_I.\ \exists v_{1S}, v_{2S}.\ v_I = (v_{1I}, v_{2I}) * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * \\ j \overset{\zeta}{\Rightarrow}_S (v_{1S}, v_{2S}) * [\mathrm{SR}]^{\pi}_{\zeta} * P_{reg}(\Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * \\ P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\![\tau_1 \times \tau_2]\!]^M(v_I, v_S) * [\mathrm{SR}]^{\pi}_{\zeta} * \\ P_{reg}(\Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * P_{reg}(\Pi, g, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array}\right\}_R$$

□

Read

Lemma .0.56 (Trade read tokens)**.**

$$\forall r, \iota, \pi.\ \boxed{\mathrm{REG}(r)}^{\iota} \vdash [\mathrm{RD}]^{\pi}_r \overset{\{\iota\}}{\Longleftrightarrow} \emptyset \; \exists h.\ locs(h, r) * toks(\pi, 1, r) * \circledast_{x \in Loc^2} [\mathrm{RD}(x)]_r$$

$$\forall r, \iota.\ \boxed{\mathrm{REG}(r)}^{\iota} \vdash [\mathrm{RD}]^{1}_r \overset{\{\iota\}}{\Longleftrightarrow}{}^{\{\iota\}} \circledast_{x \in Loc^2} [\mathrm{RD}(x)]_r$$

Lemma .0.57 (Read effect ensures well-typedness)**.**

$$\begin{array}{rl} \forall r, \phi, x, v. & effs(r, \phi, x, v) * [\mathrm{RD}(x)]_r \\ \Rightarrow & effs(r, \phi, x, v) * [\mathrm{RD}(x)]_r * v \in \phi \end{array}$$

Lemma .0.58.

$$\forall h, r, x, y, \pi. \mathit{locs}(h, r) * x \overset{\pi}{\hookrightarrow}_{I,r} y \Rightarrow \mathit{locs}(h, r) * x \overset{\pi}{\hookrightarrow}_{I,r} y * h_I(x) = y$$

Lemma .0.59.

$$\forall h, r, x, y, \pi. \mathit{locs}(h, r) * x \overset{\pi}{\hookrightarrow}_{S,r} y \Rightarrow \mathit{locs}(h, r) * x \overset{\pi}{\hookrightarrow}_{S,r} y * h_S(x) = y$$

Lemma .0.60.

$$\begin{aligned} & \forall h, r, \zeta, \pi. \\ & \quad \mathit{locs}(h, r) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \\ \Rightarrow \quad & \mathit{locs}(h, r, \{\zeta\}, \{\zeta\}) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{aligned}$$

Lemma .0.61.

$$\begin{aligned} & \forall h, h_R, r, S, \pi. \\ & \quad \mathit{locs}(h, r) * [\textsc{Im}(r, S, h_R)]^{\pi} \\ \Rightarrow \quad & \mathit{locs}(h, r, S, S) * [\textsc{Im}(r, S, h_R)]^{\pi} * h_S = h_R \end{aligned}$$

Lemma .0.62.

$$\begin{aligned} & \forall h, r, y, \zeta, S, S', \pi. \\ & \quad \mathit{locs}(h, r, S', \{\zeta\} \uplus S) \\ \Leftrightarrow \quad & \mathit{locs}(h, r, S', S) * \circledast_{(l,v) \in h_S} l \mapsto_S^{\zeta} v \end{aligned}$$

Lemma .0.63 (Implementation dereference)**.**

$$\begin{aligned} & \forall r, x, v, h, \pi. \\ & \quad \left\{\textsc{Heap} * x \overset{\pi}{\hookrightarrow}_{I,r} v * \mathit{locs}(h, r)\right\} \\ & \quad !x \\ & \quad \left\{v'.\ \textsc{Heap} * x \overset{\pi}{\hookrightarrow}_{I,r} v * \mathit{locs}(h, r) * v' = v\right\} \end{aligned}$$

Proof. By Lemma .0.58 and definition of *locs*. □

Lemma .0.64 (Specification dereference)**.**

$$\begin{aligned} & \forall h_0, h_S, e_0, e, \pi, \pi', \zeta, x, v, j. \\ & \quad \textsc{Spec}(h_0, h_S, e_0, e, \pi, \zeta) * x \mapsto_S^{\zeta} v * j \overset{\zeta}{\Rightarrow}_S !x * [\textsc{Sr}]_{\zeta}^{\pi'} \\ \Rrightarrow \quad & \textsc{Spec}(h_0, h_S, e_0, e, \pi, \zeta) * x \mapsto_S^{\zeta} v * j \overset{\zeta}{\Rightarrow}_S v * [\textsc{Sr}]_{\zeta}^{\pi'} \end{aligned}$$

Proof. $x \mapsto_S^{\zeta} v$ asserts $h_S[x \mapsto v]$. From our operational semantics we have $(h_S[x \mapsto v], !x) \rightarrow (h_S[x \mapsto v], v)$ and since we do not change the heap the update of ghost-state follows from Lemma .0.46. □

Lemma .0.65 (Specification dereference for region)**.**

$$\begin{aligned}&\forall j, x, v, r, h, h_R, \zeta, S, \pi, \pi', \pi''.\\ &\quad \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S\ !x * [\textsc{Sr}]^{\pi''}_{\zeta} * x \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v * locs(h, r) * \\ &\quad slink(r, S, h_R, \pi, \pi') * \zeta \in S\\ \Rrightarrow\quad &\textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S v * [\textsc{Sr}]^{\pi''}_{\zeta} * x \overset{\frac{1}{2}}{\hookrightarrow}_{S,r} v * locs(h, r) * \\ &\quad slink(r, S, h_R, \pi, \pi')\end{aligned}$$

Proof. By Lemma .0.59, Lemma .0.62 and Lemma .0.64. □

Lemma .0.66.

$$\begin{aligned}&\forall r, \phi, x, \zeta, S, j, h, \pi, \pi', \pi''.\\ &\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)} \vdash\\ &\left\{ j \overset{\zeta}{\Rightarrow}_S\ !x_S * [\textsc{Sr}]^{\pi''}_{\zeta} * locs(h, r) * [\textsc{Rd}(x)]_r * slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') \right\}\\ &\quad !x_I\\ &\left\{ v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi''}_{\zeta} * locs(h, r) * [\textsc{Rd}(x)]_r * \right.\\ &\qquad \left. slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') * (v_I, v_S) \in \phi \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)\}}\end{aligned}$$

Proof.

Context: $r, \phi, x, \zeta, S, j, h, \pi, \pi', \pi'', \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S\ !x_S * [\textsc{Sr}]^{\pi''}_{\zeta} * locs(h, r) * [\textsc{Rd}(x)]_r * slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)\}}$

Open $\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)$

// ▷ moved by Lemma .0.45

$$\left\{ \begin{array}{l} \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S\ !x_S * [\textsc{Sr}]^{\pi''}_{\zeta} * \exists v.\ ref(r, \triangleright \phi, x, v) * \\ locs(h, r) * [\textsc{Rd}(x)]_r * slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') \end{array} \right\}_{\emptyset}$$

$!x_I$

// Unfold *ref* and apply Lemma .0.63

$$\left\{ \begin{array}{l} v_I^2.\ \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S\ !x_S * [\textsc{Sr}]^{\pi''}_{\zeta} * \exists v.\ ref(r, \phi, x, v) * \\ locs(h, r) * [\textsc{Rd}(x)]_r * v_I = v_I^2 * slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') \end{array} \right\}_{\emptyset}$$

// Lemma .0.65

$$\left\{ \begin{array}{l} v_I^2.\ \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * \exists v.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi''}_{\zeta} * ref(r, \phi, x, v) * \\ locs(h, r) * [\textsc{Rd}(x)]_r * v_I = v_I^2 * slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') \end{array} \right\}_{\emptyset}$$

// Lemma .0.57

$$\left\{ \begin{array}{l} v_I^2.\ \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * \exists v.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi''}_{\zeta} * ref(r, \phi, x, v) * \\ (v_I^2, v_S) \in \phi * locs(h, r) * [\textsc{Rd}(x)]_r * slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') \end{array} \right\}_{\emptyset}$$

$$\left\{ \begin{array}{l} v_I^2.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S v_S * [\textsc{Sr}]^{\pi''}_{\zeta} * locs(h, r) * [\textsc{Rd}(x)]_r * \\ slink(r, \{\zeta\} \uplus S, h_S, \pi, \pi') * (v_I^2, v_S) \in \phi \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)\}}$$

□

Lemma .0.67.

$$\forall r, \zeta, \pi, \pi', h.\ [\textsc{Mu}(r, \{\zeta\})]^{\pi} \Leftrightarrow \mathit{slink}(r, \{\zeta\}, h, \pi, \pi')$$

Lemma .0.68 (Read)**.**

$$\begin{array}{l}
\forall r, \phi, x, \pi, \pi', \pi'', \pi''', j, \zeta, S, h. \\
\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)} \vdash \\
\left\{ j \overset{\zeta}{\Rightarrow}_S\ !x_S * [\textsc{Sr}]^{\pi'''}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * \mathit{slink}(r, \{\zeta\} \uplus S, h, \pi', \pi'') \right\} \\
\quad !x_I \\
\left\{ \begin{array}{l} v_I.\ \exists v_S.\ j \overset{\zeta}{\Rightarrow}_S\ v_S * [\textsc{Sr}]^{\pi'''}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * (v_I, v_S) \in \phi * \\ \quad \mathit{slink}(r, \{\zeta\} \uplus S, h, \pi', \pi'') \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rg}(r), \textsc{Rf}(x)\}}
\end{array}$$

Proof. By Lemma .0.56 and Lemma .0.66. □

Write

Lemma .0.69 (Trade write tokens)**.**

$$\begin{array}{rl}
\forall r, \iota, \pi. \boxed{\textsc{Reg}(r)}^{\iota} \vdash [\textsc{Wr}]^{\pi}_{r} & \overset{\{\iota\}}{\Longleftrightarrow}{}^{\emptyset}\ \exists h.\ \mathit{locs}(h, r) * \mathit{toks}(1, \pi, r) * \\
& \qquad \circledast_{x \in Loc^2} [\textsc{Wr}(x)]_r \\
\forall r, \iota. \boxed{\textsc{Reg}(r)}^{\iota} \vdash [\textsc{Wr}]^{1}_{r} & \overset{\{\iota\}}{\Longleftrightarrow}{}^{\{\iota\}}\ \circledast_{x \in Loc^2} [\textsc{Wr}(x)]_r
\end{array}$$

Lemma .0.70 (Assign in concrete code)**.**

$$\begin{array}{l}
\forall x, v. \\
\quad \{\textsc{Heap} * x \mapsto -\} \\
\quad\ x := v \\
\quad \{v'.\ v' = () * \textsc{Heap} * x \mapsto v\}
\end{array}$$

Lemma .0.71 (Assign in specification code)**.**

$$\begin{array}{rl}
\forall h_0, e_0, \pi, \pi', \zeta, j, e, x, v. & \\
\quad \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S\ x := v * [\textsc{Sr}]^{\pi'}_{\zeta} * x \overset{\zeta}{\mapsto}_S - & \\
\Rrightarrow \quad \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S\ () * [\textsc{Sr}]^{\pi'}_{\zeta} * x \overset{\zeta}{\mapsto}_S v &
\end{array}$$

Proof. $x \overset{\zeta}{\mapsto}_S -$ asserts $h_S[x \mapsto -]$. From the operational semantics we have $(h_S[x \mapsto -], x := v) \to (h_S[x \mapsto v], ())$ and since we do not change the domain of the heap, the update of ghost-state follows from Lemma .0.46. □

Lemma .0.72 (Exclusive ownership of region-references)**.**

$$\begin{array}{ll} & \forall r, \phi, x, v. \\ & \quad \mathit{ref}(r, \phi, x, v) * [\textsc{Wr}(x)]_r \\ \iff & \quad [\textsc{Wr}(x)]_r * x_I \overset{1}{\hookrightarrow}_{I,r} v_I * x_S \overset{1}{\hookrightarrow}_{S,r} v_S * ([\textsc{Rd}(x)]_r \vee (v \in \phi * [\textsc{NoRd}(x)]_r)) \end{array}$$

Lemma .0.73 (Update related locations with related values)**.**

$$\begin{array}{ll} & \forall r, \phi, x, v. \\ & \quad x_I \overset{1}{\hookrightarrow}_{I,r} v_I' * x_S \overset{1}{\hookrightarrow}_{S,r} v_S' * v \in \phi * ([\textsc{Rd}(x)]_r \vee (v' \in \phi * [\textsc{NoRd}(x)]_r)) \\ \Rrightarrow & \quad \mathit{ref}(r, \phi, x, v) \end{array}$$

Lemma .0.74 (Assignment)**.**

$$\begin{array}{l} \forall r, \phi, x, v, h, j, \zeta, \pi, \pi'. \\ \quad \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)} \vdash \\ \quad \left\{ j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * \mathit{locs}(h, r) * [\textsc{Wr}(x)]_r * [\textsc{Mu}(r, \{\zeta\})]^{\pi} * \phi(v) \right\} \\ \quad x_I := v_I \\ \quad \left\{ \begin{array}{l} v'.\ v' = () * j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi'}_{\zeta} * \mathit{locs}(h, r) * [\textsc{Wr}(x)]_r * \\ \quad [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)\}} \end{array}$$

Proof.

Context: $r, \phi, x, v, h, j, \zeta, \pi, \pi', \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)}, \phi(v)$

$$\left\{ j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * locs(h, r) * [\textsc{Wr}(x)]_r * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)\}}$$

Open $\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)$

$$\left\{ \begin{array}{l} \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * \textsc{Ref}(r, \phi, x) * j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ locs(h, r) * [\textsc{Wr}(x)]_r * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\emptyset}$$

// Lemma .0.72.

$$\left\{ \begin{array}{l} \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * locs(h, r) * [\textsc{Wr}(x)]_r * \\ x_I \overset{1}{\hookrightarrow}_{I,r} -\!\!* \, x_S \overset{1}{\hookrightarrow}_{S,r} -\!\!* ([\textsc{Rd}(x)]_r \vee ((-,-) \in \phi * \\ [\textsc{NoRd}(x)]_r)) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\emptyset}$$

// Lemma .0.58 and Lemma .0.59 and unfolding of *locs*

$$\left\{ \begin{array}{l} \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ \exists h'_I, h'_S.\ h_I = h'_I \uplus [x_I \mapsto -] * h_S = h'_S \uplus [x_S \mapsto -] * slink(r, \{\zeta\}, h_S, \frac{1}{2}, \frac{1}{4}) * \\ rheap_I(h_I, r) * rheap_S(h_S, r) * alloc(h, r) * \circledast_{(l,v) \in h'_I} l \mapsto v * x_I \mapsto - * \\ \circledast_{(l,v) \in h'_S} l \overset{\zeta}{\mapsto}_S v * x_S \overset{\zeta}{\mapsto}_S -\!\!* [\textsc{Wr}(x)]_r * x_I \overset{1}{\hookrightarrow}_{I,r} - * x_S \overset{1}{\hookrightarrow}_{S,r} -\!\!* \\ ([\textsc{Rd}(x)]_r \vee [\textsc{NoRd}(x)]_r) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\emptyset}$$

Frame

$$\left\{ \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * x_I \mapsto -\!\!* \, x_S \overset{\zeta}{\mapsto}_S - \right\}_{\emptyset}$$

$x_I := v_I$

$$\left\{ \begin{array}{l} v_I^1.\ v_I^1 = () * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ x_I \mapsto v_I * x_S \overset{\zeta}{\mapsto}_S - \end{array} \right\}_{\emptyset}$$

// Lemma .0.71

$$\left\{ \begin{array}{l} v_I^1.\ v_I^1 = () * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ x_I \mapsto v_I * x_S \overset{\zeta}{\mapsto}_S v_S \end{array} \right\}_{\emptyset}$$

$$\left\{ \begin{array}{l} v_I^1.\ v_I^1 = () * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ \exists h'_I, h'_S.\ h_I = h'_I[x_I \mapsto -] * h_S = h'_S[x_S \mapsto -] * slink(r, \{\zeta\}, h_S, \frac{1}{2}, \frac{1}{4}) * \\ rheap_I(h_I, r) * rheap_S(h_S, r) * alloc(h, r) * \circledast_{(l,v) \in h'_I} l \mapsto v * x_I \mapsto v_I * \\ \circledast_{(l,v) \in h'_S} l \overset{\zeta}{\mapsto}_S v * x_S \overset{\zeta}{\mapsto}_S v_S * [\textsc{Wr}(x)]_r * x_I \overset{1}{\hookrightarrow}_{I,r} -\!\!* \\ x_S \overset{1}{\hookrightarrow}_{S,r} -\!\!* ([\textsc{Rd}(x)]_r \vee [\textsc{NoRd}(x)]_r) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\emptyset}$$

// Updated region points-to by having full fraction and having both the full and the fragmental authorative parts by AFHeapUpd.

$$\left\{ \begin{array}{l} v_I^1.\ v_I^1 = () * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi'}_{\zeta} * \exists h'. \\ h' = (h_I[x_I \mapsto v_I], h_S[x_S \mapsto v_S]) * locs(h', r) * [\textsc{Wr}(x)]_r * x_I \overset{1}{\hookrightarrow}_{I,r} v_I * \\ x_S \overset{1}{\hookrightarrow}_{S,r} v_S * ([\textsc{Rd}(x)]_r \vee (\phi(v_I, v_S) * [\textsc{NoRd}(x)]_r)) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\emptyset}$$

// Lemma .0.73 and folding of Ref predicate.

$$\left\{ \begin{array}{l} v_I^1.\ \exists h', v_S^1.\ v_I^1 = () * v_S^1 = () * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S v_S^1 * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ locs(h', r) * [\textsc{Wr}(x)]_r * \textsc{Ref}(r, \phi, x) * [\textsc{Mu}(r, \{\zeta\})]^{\pi} \end{array} \right\}_{\emptyset}$$

$$\left\{ \begin{array}{l} v_I^1.\ \exists h', v_S^1.\ j \overset{\zeta}{\Rightarrow}_S v_S^1 * [\textsc{Sr}]^{\pi'}_{\zeta} * locs(h', r) * [\textsc{Wr}(x)]_r * \\ [\textsc{Mu}(r, \{\zeta\})]^{\pi} * [\![\mathbf{1}]\!]^M(v_I^1, v_S^1) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rf}(x)\}}$$

□

Lemma .0.75 (Write)**.**

$$\begin{array}{l}\forall r, \phi, x, \zeta, j, \pi, \pi', v.\\ \quad \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, \boxed{\textsc{Ref}(r, \phi, x)}^{\textsc{Rf}(x)} \vdash\\ \quad \left\{ j \overset{\zeta}{\Rightarrow}_S x_S := v_S * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\pi} * [\textsc{Wr}]^{\pi}_{r} * \phi(v) \right\}\\ \qquad x_I := v_I\\ \quad \left\{ (). \; j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Mu}(r, \{\zeta\})]^{\pi} * [\textsc{Wr}]^{\pi}_{r} * [\![\mathbf{1}]\!]^{M}((),()) \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rg}(r), \textsc{Rf}(x)\}}\end{array}$$

Proof. By Lemma .0.69 and Lemma .0.74. □

Allocate

Lemma .0.76 (New location in disjoint domain)**.**

$$\begin{array}{rl} & \forall v, h_0, h, \zeta.\\ & \quad heap_S(h, \zeta) * disj_H(h_0, h)\\ \Rrightarrow & \exists h', x.\; h' = h \uplus [x \mapsto (1, v)] * heap_S(h', \zeta) * disj_H(h_0, h') * x \overset{\zeta}{\mapsto}_S v \end{array}$$

Proof.

$$\begin{array}{rl} & heap_S(h, \zeta) * disj_H(h_0, h)\\ (\text{unfold}) \Rightarrow & \exists h_Y.\; heap_S(h, \zeta) * [h_Y]_H \wedge \mathrm{dom}(h_0) \cap h_Y = \emptyset \wedge\\ & (\mathrm{dom}(h) \setminus \mathrm{dom}(h_0)) \subset h_Y\\ (\text{below}) \Rightarrow & \exists h_Y, x.\; heap_S(h, \zeta) * [h_Y]_H \wedge \mathrm{dom}(h_0) \cap h_Y = \emptyset \wedge\\ & (\mathrm{dom}(h) \setminus \mathrm{dom}(h_0)) \subset h_Y * x \notin \mathrm{dom}(h) * x \in \mathrm{dom}(h_Y)\\ (\text{rewrite}) \Rightarrow & \exists h_Y, x, h'.\; h' = h \uplus [x \mapsto (1, v)] * heap_S(h, \zeta) * [h_Y]_H \wedge\\ & \mathrm{dom}(h_0) \cap h_Y = \emptyset \wedge (\mathrm{dom}(h') \setminus \mathrm{dom}(h_0)) \subset h_Y * x \notin \mathrm{dom}(h)\\ (\text{fold}) \Rightarrow & \exists x, h'.\; h' = h \uplus [x \mapsto (1, v)] * heap_S(h, \zeta) * disj_H(h_0, h')\\ (\textsc{FpAlloc}) \Rrightarrow & \exists x, h'.\; h' = h \uplus [x \mapsto (1, v)] * heap_S(h', \zeta) * disj_H(h_0, h') *\\ & x \overset{\zeta}{\mapsto}_S v \end{array}$$

From h_Y being enumerable and $\mathrm{dom}(h)$ being finite, we can pick an x such that $x \notin \mathrm{dom}(h)$ and $x \in \mathrm{dom}(h_Y)$.

□

Lemma .0.77 (Trade allocate token)**.**

$$\begin{array}{c} \forall h, r, \pi.\; alloc(h, r) * [\textsc{Al}]^{\pi}_{r} \Leftrightarrow [\textsc{Al}]^{\pi}_{r} * [\textsc{Al}(h_I, h_S)]^{1}_{r}\\ \forall h, r.\; alloc(h, r) * [\textsc{Al}]^{1}_{r} \Lleftarrow\Rrightarrow alloc(h, r) * [\textsc{Al}(h_I, h_S)]^{\frac{1}{2}}_{r} \end{array}$$

Lemma .0.78 (Allocate in concrete code)**.**

$$\begin{array}{l}\forall x, v.\\ \quad \{\textsc{Heap}\}\\ \quad\quad \textit{new}\ v\\ \quad \{l.\ \textsc{Heap} * l \mapsto v\}\end{array}$$

Lemma .0.79 (Allocate in specification code)**.**

$$\begin{array}{rl} & \forall e_0, h_0, j, x, v, \zeta, \pi.\\ & \quad \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S \textit{new}\ v\\ \Rrightarrow & \quad \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S () * [\textsc{Sr}]^{\pi}_{\zeta} * \exists x.\ x \overset{\zeta}{\mapsto}_S v\end{array}$$

Proof.

$$\begin{array}{rl} & \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S \mathsf{new}\ v\\ \Rrightarrow & \exists h, e, \pi'.\ \textsc{Spec}(h_0, h, e_0, e, \pi', \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S \mathsf{new}\ v\\ \Rrightarrow & \exists h, e, \pi'.\ heap_S(h, \zeta) * mctx(e, \zeta) * (h_0, e_0) \rightarrow^* (h, e) *\\ & \quad [\textsc{Sr}]^{\pi'+\pi}_{\zeta} * disj_H(h_0, h) * j \overset{\zeta}{\Rightarrow}_S \mathsf{new}\ v\\ (\textit{Lemma}\ .0.76)\Rrightarrow & \exists h, h', e, \pi'.\ heap_S(h', \zeta) * mctx(e, \zeta) * (h_0, e_0) \rightarrow^* (h, e) *\\ & \quad [\textsc{Sr}]^{\pi'+\pi}_{\zeta} * disj_H(h_0, h') * j \overset{\zeta}{\Rightarrow}_S \mathsf{new}\ v * x \overset{\zeta}{\mapsto}_S v *\\ & \quad h' = h \uplus [x \mapsto (1, v)]\\ (\textit{Lemma}\ .0.46)\Rrightarrow & \exists h', e', \pi'.\ heap_S(h', \zeta) * mctx(e, \zeta) * (h_0, e_0) \rightarrow^* (h', e') *\\ & \quad [\textsc{Sr}]^{\pi'+\pi}_{\zeta} * disj_H(h_0, h') * j \overset{\zeta}{\Rightarrow}_S () * x \overset{\zeta}{\mapsto}_S v\\ (\text{fold})\Rrightarrow & \exists h', e', \pi'.\ \textsc{Spec}(h_0, h', e_0, e', \pi', \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S () * x \overset{\zeta}{\mapsto}_S v\\ (\text{fold})\Rightarrow & \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S () * x \overset{\zeta}{\mapsto}_S v\end{array}$$

We can take the step $(h, \mathsf{new}\ v) \rightarrow (h'[x \mapsto v], ())$ since we have $x \notin \mathrm{dom}(h)$. □

Lemma .0.80 (Extending region heap)**.**

$$\begin{array}{rl} & \forall x, v, \pi, r, \pi, \iota.\ \boxed{\textsc{Reg}(r)}^{\iota}\\ \vdash & x_I \mapsto v_I * x_S \overset{\zeta}{\mapsto}_S v_S * [\textsc{Al}]^{\pi}_{r} * [\textsc{Mu}(r, \{\zeta\})]^{\pi}\\ \{i\}\Rrightarrow^{\{i\}} & x_I \overset{1}{\hookrightarrow}_{I,r} v_I * x_S \overset{1}{\hookrightarrow}_{S,r} v_S * [\textsc{NoRd}(x)]_r * [\textsc{Al}]^{\pi}_{r} * [\textsc{Mu}(r, \{\zeta\})]^{\pi}\end{array}$$

Proof. By VSInv we obtain $\triangleright(\exists h.\, locs(h,r) * toks(1,1,r))$ and we can remove the later by Lemma .0.45. By having $locs(h,r)$, $x_I \mapsto v_I$ and $x_S \mapsto^{\zeta}_{S} v_S$ it is the case $x_I \notin \text{dom}(h_I)$ and $x_S \notin \text{dom}(h_S)$. By AFHeapAdd we obtain $x_I \overset{1}{\hookrightarrow}_{I,r} v_I$ and $x_S \overset{1}{\hookrightarrow}_{S,r} v_S$. By Lemma .0.77 we obtain the exclusive token guarding the domains of h_I and h_S and we can do a frame-preserving update and we also obtain $[\text{NoRd}(x)]_r$. We can fold $\exists h'.\, locs(h',r)$ since we have provided all spec points to required by *slink*, which we know since we own $[\text{Mu}(r,\{\zeta\})]^{\pi}$. □

Lemma .0.81 (Allocating region reference)**.**

$$\forall x, v, \phi, r.$$
$$x_I \overset{1}{\hookrightarrow}_{I,r} v_I * x_S \overset{1}{\hookrightarrow}_{S,r} v_S * v \in \phi * [\text{NoRd}(x)]_r$$
$$\Rightarrow^{\{\text{Rf}(x)\}}_{\emptyset} \quad \boxed{\text{Ref}(r,\phi,x)}^{\text{Rf}(x)}$$

Proof.

$$x_I \overset{1}{\hookrightarrow}_{I,r} v_I * x_S \overset{1}{\hookrightarrow}_{S,r} v_s * v \in \phi * [\text{NoRd}(x)]_r$$
$$\Longleftrightarrow \quad x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * x_I \overset{\frac{1}{2}}{\hookrightarrow}_{I,r} v_I * effs(r,\phi,x,v)$$
$$\Rrightarrow \quad ref(r,\phi,x,v)$$
$$\Rrightarrow^{\{\text{Rf}(x)\}} \quad \boxed{\text{Ref}(r,\phi,x)}^{\text{Rf}(x)}$$

□

Lemma .0.82 (Allocate)**.**

$$\forall r, \zeta, j, v, \phi, \pi, \pi', \pi''.\ \boxed{\text{Heap}}^{\text{Hp}}, \boxed{\text{Spec}(h_0, e_0, \zeta)}^{\text{Sp}(\zeta)}, \boxed{\text{Reg}(r)}^{\text{Rg}(r)} \vdash$$
$$\left\{ j \overset{\zeta}{\Rightarrow}_S \textbf{\textit{new}}\, v_S * [\text{Sr}]^{\pi''}_{\zeta} * [\text{Al}]^{\pi}_{r} * [\text{Mu}(r,\{\zeta\})]^{\pi'} * v \in \phi \right\}$$
$$\textbf{\textit{new}}\, v_I$$
$$\left\{ \begin{array}{l} l_I.\ \exists l_S.\ j \overset{\zeta}{\Rightarrow}_S l_S * [\text{Sr}]^{\pi''}_{\zeta} * [\text{Al}]^{\pi}_{r} * [\text{Mu}(r,\{\zeta\})]^{\pi'} * \\ \quad \boxed{\text{Ref}(r,\phi,(l_I,l_S))}^{\text{Rf}(l_I,l_S)} \end{array} \right\}_{\{\text{Hp},\text{Sp}(\zeta),\text{Rg}(r)\}}$$

Proof.

$$
\begin{array}{l}
\text{Context } r,\zeta,j,v,\phi,\pi,\pi',\pi'',\boxed{\textsc{Heap}}^{\textsc{Hp}},\boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)},\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \\
\left\{ j \overset{\zeta}{\Rightarrow}_S \textsf{new}\ v_S * [\textsc{Sr}]^{\pi''}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\pi'} * v \in \phi \right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta),\textsc{Rg}(r)\}} \\
\text{Open } \textsc{Hp},\textsc{Sp}(\zeta) \\
\quad \left\{ \textsc{Heap} * \textsc{Spec}(h_0,e_0,\zeta) * j \overset{\zeta}{\Rightarrow}_S \textsf{new}\ v_S * [\textsc{Sr}]^{\pi''}_{\zeta} \right\}_{\{\textsc{Rg}(r)\}} \\
\quad\ \textsf{new}\ v_I \\
\quad \texttt{// Lemma .0.78} \\
\quad \left\{ l_I.\ \textsc{Heap} * l_I \mapsto v_I * \textsc{Spec}(h_0,e_0,\zeta) * j \overset{\zeta}{\Rightarrow}_S \textsf{new}\ v_S \right\}_{\{\textsc{Rg}(r)\}} \\
\quad \texttt{// Lemma .0.79} \\
\quad \left\{ \begin{array}{l} l_I.\ \exists l_S.\ \textsc{Heap} * l_I \mapsto v_I * \textsc{Spec}(h_0,e_0,\zeta) * j \overset{\zeta}{\Rightarrow}_S l_S * [\textsc{Sr}]^{\pi''}_{\zeta} * \\ l_S \mapsto^{\zeta}_{S} v_S \end{array} \right\}_{\{\textsc{Rg}(r)\}} \\
\left\{ \begin{array}{l} l_I.\ \exists l_S.\ j \overset{\zeta}{\Rightarrow}_S l_S * l_I \mapsto v_I * l_S \mapsto^{\zeta}_{S} v_S * [\textsc{Sr}]^{\pi''}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \\ [\textsc{Mu}(r,\{\zeta\})]^{\pi'} * v \in \phi \end{array} \right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta),\textsc{Rg}(r)\}} \\
\texttt{// Lemma .0.80 and Lemma .0.81} \\
\left\{ \begin{array}{l} l_I.\ \exists l_S.\ j \overset{\zeta}{\Rightarrow}_S l_S * [\textsc{Sr}]^{\pi''}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * [\textsc{Mu}(r,\{\zeta\})]^{\pi'} * \\ \boxed{\textsc{Ref}(r,\phi,(l_I,l_S))}^{\textsc{Rf}(l_I,l_S)} \end{array} \right\}_{\{\textsc{Hp},\textsc{Sp}(\zeta),\textsc{Rg}(r)\}}
\end{array}
$$

□

Masking

Lemma .0.83.

$$
\begin{array}{c}
\forall \Pi,\Lambda,\varepsilon,M_1,M_2,\zeta,g.\ (\forall \rho \in \Pi,\Lambda.\ M_1(\rho) = M_2(\rho)) \Rightarrow \\
P_{reg}(\Lambda,\mathbf{1},\varepsilon,M_1,\zeta) * P_{reg}(\Pi,g,\varepsilon,M_1,\zeta) = \\
P_{reg}(\Lambda,\mathbf{1},\varepsilon,M_2,\zeta) * P_{reg}(\Pi,g,\varepsilon,M_2,\zeta)
\end{array}
$$

Proof. Unfolding shows syntactic equality between ghost-resources. □

Lemma .0.84.

$$
\begin{array}{c}
\forall \Pi,\Lambda,M_1,M_2,e,\phi,\psi,\varepsilon.\ (\forall \rho \in \Pi,\Lambda.\ M_1(\rho) = M_2(\rho) \wedge \phi = \psi) \Rightarrow \\
\mathcal{E}^{\Pi,\Lambda,M_1}_{\varepsilon,M}(\phi)(e) = \mathcal{E}^{\Pi,\Lambda,M_2}_{\varepsilon,M}(\psi)(e)
\end{array}
$$

Proof. Follows by Lemma .0.83 and by $\phi = \psi$. □

Lemma .0.85.

$$
\forall \tau,M_1,M_2.\ (\forall \rho \in FRV(\tau).\ M_1(\rho) = M_2(\rho)) \Rightarrow [\![\tau]\!]^{M_1} = [\![\tau]\!]^{M_2}
$$

Proof. Induction on τ. The simple types are straight forward even for the binary case. Arrow type follows by Lemma .0.84. To remind the reader, the following is the definition of reference types:

$$[\![\mathsf{ref}_\rho\ \tau]\!]^M \triangleq \lambda x.\ \boxed{\textsc{Ref}(M(\rho), [\![\tau]\!]^M, x)}^{\textsc{Rf}(x)} * \boxed{\textsc{Reg}(M(\rho))}^{\textsc{Rg}(M(\rho))}$$

From $M_1(\rho) = M_2(\rho)$ we have $\boxed{\textsc{Reg}(M_1(\rho))}^{\textsc{Rg}(M_1(\rho))} = \boxed{\textsc{Reg}(M_2(\rho))}^{\textsc{Rg}(M_2(\rho))}$. Similarly, we have to show:

$$\boxed{\textsc{Ref}(M_1(\rho), [\![\tau]\!]^M, x)}^{\textsc{Rf}(x)} = \boxed{\textsc{Ref}(M_2(\rho), [\![\tau]\!]^M, x)}^{\textsc{Rf}(x)}$$

which follows directly from $M_1(\rho) = M_2(\rho)$ and the induction hypothesis. □

Lemma .0.86 (Creating monoids)**.**

$$\top \Rrightarrow \exists r.\ locs(\emptyset, r) * toks(1, 1, r) * r \notin \text{dom}(M)$$

Proof. Follows by repeated application of NewGhost. □

Lemma .0.87.

$$\top \Rrightarrow \exists r.\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * [\textsc{Al}]^1_r$$

Proof. Follows by Lemma .0.86 and NewInv for creating $\exists r.\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}$. □

Soundness

Definition .0.88. $\Pi \mid \Lambda \mid \Gamma \vdash e_1 \leq_{ctx} e_2 : \tau, \varepsilon$ *iff for all contexts* C*, values* v*, and heaps* h_1 *such that* $C : (\Pi \mid \Lambda \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (- \mid - \mid - \vdash \boldsymbol{B}, \emptyset)$ *and* $[]; C[e_1] \to^* h_1; v$ *there exists a heap* h_2 *such that* $[]; C[e_2] \to^* h_2; v$*.*

Theorem .0.89 (Iris soundness)**.** *For all* $p \in Props$*,* $e \in Exp$*,* $q : Val \to Props$*,* $n, k \in \mathbb{N}$*,* $v \in Val$*,* $r \in Res$*,* $\sigma, \sigma' \in State$*,* $W \in World$*, and* $\mathcal{E} \in Mask$*, if*

$$valid(\{p\}\ e\ \{q\}_{\mathcal{E}}) \qquad e, \sigma \to^n v, \sigma' \qquad (n + k + 1, r) \in p(W)$$
$$(n + k + 1, \sigma) \in \lfloor r \rfloor^W_{\mathcal{E}}$$

then there exists a $W' \geq W$ *and* $r' \in Res$ *such that*

$$(k + 1, r') \in q(v)(W') \qquad (k + 1, \sigma') \in \lfloor r' \rfloor^{W'}_{\mathcal{E}}$$

Lemma .0.90. *If* $\Pi \mid \Lambda \mid \Gamma \models_{\text{PAR}} e_1 \leq_{log} e_2 : \tau, \varepsilon$ *and* $C : (\Pi \mid \Lambda \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (\Pi' \mid \Lambda' \mid \Gamma' \vdash \tau', \varepsilon')$ *then* $\Pi' \mid \Delta' \mid \Gamma' \models_{\text{PAR}} C[e_1] \leq_{log} C[e_2] : \tau', \varepsilon'$*.*

Lemma .0.91. *If* $- \mid - \mid - \models_{\text{PAR}} e_1 \leq_{log} e_2 : \tau, \varepsilon$ *then*

$$\vdash \{\top\}\ e_1\ \{\lambda v_1.\, \exists h_2.\, \exists v_2.\, (v_1, v_2) \in [\![\tau]\!] * []; e_2 \to^* h_2; v_2\}$$

Proof.

$\{\top\}$
$\left\{\exists \zeta.\ \boxed{\textsc{Spec}([], e_2, \zeta)}^{\textsc{Sp}(\zeta)} * 0 \overset{\zeta}{\Rightarrow}_S e_2\right\}$
e_1
$\left\{v_1.\exists \zeta.\ \boxed{\textsc{Spec}([], e_2, \zeta)}^{\textsc{Sp}(\zeta)} * \exists v_2.\ [\![\tau]\!](v_1, v_2) * 0 \overset{\zeta}{\Rightarrow}_S v_2\right\}$
$\{v_1.\exists v_2.\ [\![\tau]\!](v_1, v_2) * \exists h.\ []; e_2 \to^* h; v_2\}$

□

Lemma .0.92. *If* $- \mid - \mid - \models_{\textsc{Par}} e_1 \leq_{log} e_2 : \mathbf{B}, \varepsilon$ *and* $[]; e_1 \to^* h_1; v_1$ *then there exists an* h_2 *such that* $[]; e_2 \to^* h_2; v_1$.

Proof.

- from the $- \mid - \mid - \vdash e_1 \leq_{log} e_2 : \mathbf{B}, \varepsilon$ assumption it follows by Lemma .0.91 that

$$\vdash \{\top\}\ e_1\ \{\lambda v_1.\exists h_2.\exists v_2.\ v_1 = v_2 * []; e_2 \to^* h_2; v_2\}$$

- hence, by Theorem .0.89, it follows that there exists W and r such that

$$(2, r') \in (\lambda v_1.\exists h_2.\exists v_2.\ v_1 = v_2 * []; e_2 \to^* h_2; v_2)(v_1)(W)$$

  and $(2, h_I) \in \lfloor r' \rfloor_{\mathcal{E}}^{W}$

- hence, there exists v_2, h_2 such that $v_1 = v_2$ and $[]; e_2 \to^* h_2; v_2$.

□

Theorem .0.93 (Soundness of $\text{LR}_{\textsc{Par}}$)**.** *If* $\Pi \mid \Delta \mid \Gamma \models_{\textsc{Par}} e_1 \leq_{log} e_2 : \tau, \varepsilon$ *then* $\Pi \mid \Delta \mid \Gamma \vdash e_1 \leq_{ctx} e_2 : \tau, \varepsilon$

Proof.

- let $C : (\Pi \mid \Delta \mid \Gamma \vdash \tau, \varepsilon) \rightsquigarrow (- \mid - \mid - \vdash \mathbf{B}, \emptyset)$ and assume that $[]; C[e_1] \to^* h_1; v$
- by Lemma .0.90 it follows that $- \mid - \mid - \vdash C[e_1] \leq_{log} C[e_2] : \mathbf{B}, \emptyset$
- and thus, by Lemma .0.92, there exists h_2 such that $[]; C[e_1] \to^* h_2; v$

□

Effect-Dependent Transformations

Parallelization

Theorem .0.94 (Parallelization)**.** *Assuming*

1. $\Lambda_3 \mid \Lambda_1 \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1$
2. $\Lambda_3 \mid \Lambda_2 \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2$
3. $\mathsf{als}\ \varepsilon_1 \cup \mathsf{wrs}\ \varepsilon_1 \subseteq \Lambda_1$ *and* $\mathsf{als}\ \varepsilon_2 \cup \mathsf{wrs}\ \varepsilon_2 \subseteq \Lambda_2$
4. $\mathsf{rds}\ \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ *and* $\mathsf{rds}\ \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then

$$\Pi \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma \vdash e_1 \| e_2 \preceq (e_1, e_2) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$$

The two next lemmas provides the base of the proof:

Lemma .0.95 (Framed heap)**.** *If for all heaps* h, h', h_F *and expression* e, e'*:*

$$(h; e) \rightarrow^* (h'; e) \wedge h_F \# h \wedge h_f \# h'$$

then

$$(h_F \uplus h; e) \rightarrow^* (h_F \uplus h'; e')$$

Proof. By induction. □

Lemma .0.96 (New disjoint range)**.**

$$\forall f, g, h.\, disj_H(f, g) \Rrightarrow disj_H(f, g) * disj_H(h, h,)$$

Lemma .0.97 (*disjoint* ensures disjointness)**.**

$$\forall f_1, f_2, g, h, Z.\ disj_H(f_1, g \uplus f_2) * disj_H(f_2, h) \Rrightarrow disj_H(f_1, g \uplus h)$$

We define the following short-hand notations:

$$I(R) \triangleq \{\textsc{Rg}(r) \mid r \in R\}$$

$$HRef(h, r) \triangleq \exists S.\, locs(h, r, S, \emptyset) * toks(1, 1, r)$$

$$heaps(S, h) \triangleq \circledast_{\zeta \in S} \circledast_{(l,v) \in h} l \mapsto_S^{\zeta} v$$

$$P_f(\Lambda, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) \triangleq$$
$$h_3 = \uplus_{r \in M(\Lambda)} h_{3R}(r) * heaps(\{\zeta\}, h_1 \uplus h_2 \uplus h_3) *$$
$$\circledast_{\rho \in \mathsf{rds}\ \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2} [\textsc{Rd}]^{\frac{1}{2}}_{M(\rho)} * \circledast_{r \in M(\Lambda)} [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h_{3R}(r))]^{\frac{1}{4}}$$

Lemma .0.98.

$$\forall r, S, \pi, \pi', r, y. \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash$$
$$slink(r, S, y, \pi, \pi') \overset{\{\textsc{Rg}(r)\}}{\Rrightarrow}{}^{\emptyset} \exists h. HRef(h, r) * slink(r, S, h, \pi, \pi') * heaps(S, h)$$

Lemma .0.99.

$$\forall r, S, r. \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} \vdash$$
$$slink(r, S, h, \tfrac{1}{2}, \tfrac{3}{4}) * HRef(h, r) * heaps(S', h) \overset{\emptyset}{\Rrightarrow}{}^{\{\textsc{Rg}(r)\}} slink(r, S', h, \tfrac{1}{2}, \tfrac{3}{4})$$

Lemma .0.100 (Create branching specification invariant)**.**

$$\begin{array}{ll} & \forall h, e. \\ & \quad disj_H(h, h) \\ \Rrightarrow & \exists \zeta. \textsc{Spec}(h, e, \zeta) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta} * 0 \overset{\zeta}{\Rrightarrow}_S e * heaps(\{\zeta\}, h) \end{array}$$

Lemma .0.101 (Prepare None-interference parallelization)**.**

$$\begin{array}{l} \forall j, e_1, e_2, \Lambda_1, \Lambda_2, \Lambda_3, \varepsilon_1, \varepsilon_2, M, \zeta, h_0, h_S, T_0, T.\ R = I(M(\Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3)) \Rrightarrow \\ \quad P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \{\zeta\}) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \{\zeta\}) * \\ \quad P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \{\zeta\}) * disj_H(h_0, h_S) \\ \overset{R}{\Rrightarrow}{}^{R \cup \{\textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}} \\ \quad \exists \zeta_1, \zeta_2, h_1, h_2, h_3, h_{3R}.\ S(\zeta_1, 0, h_1 \uplus h_3, e_1, e_1, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{\mathbf{1}}{2}, \varepsilon_1, M) * \\ \quad S(\zeta_2, 0, h_2 \uplus h_3, e_2, e_2, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{\mathbf{1}}{2}, \varepsilon_2, M) * disj_H(h_0, h_S) * \\ \quad P_f(\Lambda_3, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) \end{array}$$

Proof.

$$\left\{\begin{array}{l} P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \{\zeta\}) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \{\zeta\}) * P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \{\zeta\}) * \\ disj_H(h_0, h_S) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} P_{regs}(\Lambda_1 \cup \Lambda_2 \cup \Lambda_3, M) * P_{effs}(\Lambda_1, \mathbf{1}, \varepsilon_1, M) * P_{effs}(\Lambda_2, \mathbf{1}, \varepsilon_2, M) * \\ P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) * P_{par}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \{\zeta\}) * \\ P_{par}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \{\zeta\}) * P_{par}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \{\zeta\}) * disj_H(h_0, h_S) \end{array}\right\}_R$$

Frame

$$\left\{\begin{array}{l} P_{par}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \{\zeta\}) * P_{par}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \{\zeta\}) * \\ P_{par}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \{\zeta\}) * disj_H(h_0, h_S) \end{array}\right\}_R$$

$$\left\{\circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} [\textsc{Mu}(M(\rho), \{\zeta\})]^{\frac{1}{2}} * disj_H(h_0, h_S)\right\}_R$$

// Lemma .0.98

$$\left\{\begin{array}{l} \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} \exists h.\, HRef(h, M(\rho)) * [\textsc{Mu}(M(\rho), \{\zeta\})]^{\frac{1}{2}} * heaps(\{\zeta\}, h) * \\ disj_H(h_0, h_S) \end{array}\right\}_\emptyset$$

$$\left\{\begin{array}{l} \exists h.\, \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} HRef(h(\rho), M(\rho)) * [\textsc{Mu}(M(\rho), \{\zeta\})]^{\frac{1}{2}} * heaps(\{\zeta\}, h(\rho)) * \\ disj_H(h_0, h_S) \end{array}\right\}_\emptyset$$ Let

$h_i = \prod_{\rho \in \Lambda_i} h(\rho)$ for $i \in \{1, 2, 3\}$

// Follows from Lemma .0.96

$$\left\{\begin{array}{l} \exists h.\, \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} HRef(h(\rho), M(\rho)) * [\textsc{Mu}(M(\rho), \{\zeta\})]^{\frac{1}{2}} * heaps(\{\zeta\}, h(\rho)) * \\ disj_H(h_0, h_S) * disj_H(h_1 \uplus h_3, h_1 \uplus h_3) * disj_H(h_2 \uplus h_3, h_2 \uplus h_3) \end{array}\right\}_\emptyset$$

// Follows from Lemma .0.100

$$\left\{\begin{array}{l} \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} HRef(h(\rho), M(\rho)) * [\textsc{Mu}(M(\rho), \{\zeta\})]^{\frac{1}{2}} * heaps(\{\zeta\}, h(\rho)) * \\ disj_H(h_0, h_S) \exists \zeta_1.\, \textsc{Spec}(h_1 \uplus h_3, e_1, \zeta_1) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_1} * 0 \overset{\zeta_1}{\Longrightarrow}_S e_1 * \\ heaps(\{\zeta_1\}, h_1 \uplus h_3) * \exists \zeta_2.\, \textsc{Spec}(h_2 \uplus h_3, e_2, \zeta_2) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_2} * 0 \overset{\zeta_2}{\Longrightarrow}_S e_2 * \\ heaps(\{\zeta_2\}, h_2 \uplus h_3) \end{array}\right\}_\emptyset$$

Let $E(\zeta_1, \zeta_2) = \textsc{Spec}(h_1 \uplus h_3, e_1, \zeta_1) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_1} * 0 \overset{\zeta_1}{\Longrightarrow}_S e_1 *$

$\textsc{Spec}(h_2 \uplus h_3, e_2, \zeta_2) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_2} * 0 \overset{\zeta_2}{\Longrightarrow}_S e_2$

$$\left\{\begin{array}{l} \exists \zeta_1, \zeta_2.\, E(\zeta_1, \zeta_2) * disj_H(h_0, h_S) * \circledast_{\rho \in \Lambda_1} [\textsc{Mu}(M(\rho), \{\zeta_1\})]^{\frac{1}{2}} * \\ \circledast_{\rho \in \Lambda_2} [\textsc{Mu}(M(\rho), \{\zeta_2\})]^{\frac{1}{2}} * \circledast_{\rho \in \Lambda_3} [\textsc{Im}(M(\rho), \{\zeta_1, \zeta_2\}, h(\rho))]^{\frac{3}{4}} * \\ \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} heaps(\{\zeta\}, h(\rho)) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} \exists \zeta_1, \zeta_2.\, E(\zeta_1, \zeta_2) * disj_H(h_0, h_S) * P_{regs}((\Lambda_1, \Lambda_2, \Lambda_3), M) * P_{effs}(\Lambda_1, \mathbf{1}, \varepsilon_1, M) * \\ P_{effs}(\Lambda_1, \mathbf{1}, \varepsilon_2, M) * P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) * \circledast_{\rho \in \Lambda_1} [\textsc{Mu}(M(\rho), \{\zeta_1\})]^{\frac{1}{2}} * \\ \circledast_{\rho \in \Lambda_2} [\textsc{Mu}(M(\rho), \{\zeta_2\})]^{\frac{1}{2}} * \circledast_{\rho \in \Lambda_3} [\textsc{Im}(M(\rho), \{\zeta_1, \zeta_2\}, h(\rho))]^{\frac{3}{4}} * \\ \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} heaps(\{\zeta\}, h(\rho)) \end{array}\right\}_R$$

$$\left\{\begin{array}{l} \exists \zeta_1, \zeta_2.\, E(\zeta_1, \zeta_2) * disj_H(h_0, h_S) * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \{\zeta_1\}) * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \{\zeta_2\}) * P_{reg}(\Lambda_3, \frac{1}{4}, \varepsilon_1, M, \{\zeta_1\}) * P_{reg}(\Lambda_3, \frac{1}{4}, \varepsilon_2, M, \{\zeta_2\}) * \\ \circledast_{\rho \in \Lambda_3} [\textsc{Im}(M(\rho), \{\zeta_1, \zeta_2\}, h(\rho))]^{\frac{1}{4}} * \circledast_{\rho \in \Lambda_1, \Lambda_2, \Lambda_3} heaps(\{\zeta\}, h(\rho)) * \\ \circledast_{\rho \in \Lambda_3 \cap (\mathsf{rds}\,((\varepsilon_1 \cup \varepsilon_2) \setminus (\varepsilon_1 \cap \varepsilon_2)))} [\textsc{Rd}]^{\frac{1}{2}}_{M(\rho)} \end{array}\right\}_R$$

$$\left\{\begin{array}{l}\exists\zeta_1,\zeta_2.\ disj_H(h_0,h_S) * S(\zeta_1,0,h_1,e_1,e_1,\frac{1}{2},(\Lambda_1,\Lambda_3),\mathbf{\frac{1}{2}},\varepsilon_1,M) * \\ S(\zeta_2,0,h_2,e_2,e_2,\frac{1}{2},(\Lambda_2,\Lambda_3),\mathbf{\frac{1}{2}},\varepsilon_2,M) * \\ \circledast_{\rho\in\Lambda_3}[\textsc{Im}(M(\rho),\{\zeta_1,\zeta_2\},h(\rho))]^{\frac{1}{4}} * \circledast_{\rho\in\Lambda_1,\Lambda_2,\Lambda_3} heaps(\{\zeta\},h(\rho)) * \\ \circledast_{\rho\in\Lambda_3\cap(rds\ ((\varepsilon_1\cup\varepsilon_2)\setminus(\varepsilon_1\cap\varepsilon_2)))}[\textsc{Rd}]^{\frac{1}{2}}_{M(\rho)}\end{array}\right\}_{R\cup\{\textsc{Sp}(\zeta_1),\textsc{Sp}(\zeta_2)\}}$$

$$\left\{\begin{array}{l}S(\zeta_1,0,h_1,e_1,e_1,\frac{1}{2},(\Lambda_1,\Lambda_3),\mathbf{\frac{1}{2}},\varepsilon_1,M) * \\ S(\zeta_2,0,h_2e_2,e_2,\frac{1}{2},(\Lambda_2,\Lambda_3),\mathbf{\frac{1}{2}},\varepsilon_2,M) * \\ P_f(\Lambda_3,M,\zeta,\zeta_1,\zeta_2,\varepsilon_1,\varepsilon_2,h_1,h_2,h_3,h) * disj_H(h_0,h_S)\end{array}\right\}_{R\cup\{\textsc{Sp}(\zeta_1),\textsc{Sp}(\zeta_2)\}}$$

□

Lemma .0.102 (Combine shared part with frame)**.**

$$\begin{array}{ll}&\forall\Lambda,\varepsilon_1,\varepsilon_2,M,\zeta_1,\zeta_2,h.\\ &\quad(\mathsf{wrs}\,(\varepsilon_1\cup\varepsilon_2)\cup\mathsf{als}\,(\varepsilon_1\cup\varepsilon_2))\cap\Lambda=\emptyset\Rightarrow\\ &\quad P_{reg}(\Lambda,\mathbf{\frac{1}{2}},\varepsilon_1,M,\zeta_1) * P_{reg}(\Lambda,\mathbf{\frac{1}{2}},\varepsilon_2,M,\zeta_2) * \circledast_{\rho\in\mathsf{rds}\,\varepsilon_1\cup\varepsilon_2\setminus\varepsilon_1\cap\varepsilon_2}[\textsc{Rd}]^{\frac{1}{2}}_{M(\rho)} *\\ &\quad\circledast_{r\in M(\Lambda)}[\textsc{Im}(r,\{\zeta_1,\zeta_2\},h(r))]^{\frac{1}{4}}\\ \Rrightarrow\quad&\circledast_{r\in M(\Lambda)}[\textsc{Im}(r,\{\zeta_1,\zeta_2\},h(r))]^{\frac{3}{4}} * P_{effs}(\Lambda,\mathbf{1},\varepsilon_1\cup\varepsilon_2,M) * P_{regs}(\Lambda,M)\end{array}$$

Lemma .0.103.

$$\begin{array}{ll}&\forall\zeta,j,e_0,e,h,h_0.\ \boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)}\vdash\\ &\quad j\overset{\zeta}{\Rightarrow}_S e * heaps(h,\{\zeta\})\\ {\scriptstyle\{\textsc{Sp}(\zeta)\}}\Rrightarrow^{\emptyset}\quad&\exists h_S,e'.\,\textsc{Spec}(h_0,h\uplus h_S,e_0,e',\tfrac{1}{2},\zeta) * j\overset{\zeta}{\Rightarrow}_S e * heaps(h,\{\zeta\})\end{array}$$

Lemma .0.104 (Frozen regions are frames)**.**

$$\begin{array}{l}\forall h,h_f,\zeta,r,\pi.\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)},\zeta\in S\vdash\\ \quad heap_S(h,\zeta) * [\textsc{Im}(r,S,h_f)]^{\pi}\ {\scriptstyle\{\textsc{Rg}(r)\}}\Rrightarrow^{\emptyset}\exists h'.\ heap_S(h'\uplus h_f,\zeta) * [\textsc{Im}(r,S,h_f)]^{\pi}\end{array}$$

Proof. Follows Lemma .0.61 for each region. □

Lemma .0.105 (Obtain disjoint token by trading specification runner)**.**

$$\begin{array}{ll}&\forall h_0,h,e_0,e,\frac{1}{2},\zeta.\\ &\quad\textsc{Spec}(h_0,h,e_0,e,\tfrac{1}{2},\zeta) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta}\\ \Rightarrow\quad&\textsc{Spec}(h_0,h,e_0,e,1,\zeta) * disj_H(h_0,h) * (h_0,e_0)\rightarrow^*(h,e)\end{array}$$

Lemma .0.106 (Combining new specs with old spec)**.**

$$
\begin{array}{ll}
& \forall h_0, h_S, h_1, h_1', e_0, e, e_1, e_1', \zeta, \zeta'. \\
& \quad \textsc{Spec}(h_1, h_1', e_1, e_1', \tfrac{1}{2}, \zeta') * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta'} * \\
& \quad \textsc{Spec}(h_0, h_S \uplus h_1, e_0, e, \tfrac{1}{2}, \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S e_1 * heaps(\{\zeta\}, h_1) \\
\Rrightarrow & \quad \exists e''.\ \textsc{Spec}(h_1, h_1', e_1, e_1', 1, \zeta') * \\
& \quad \textsc{Spec}(h_0, h_S \uplus h_1', e_0, e'', \tfrac{1}{2}, \zeta) * [\textsc{Sr}]^{\pi}_{\zeta} * j \overset{\zeta}{\Rightarrow}_S e_1' * heaps(\{\zeta\}, h_1')
\end{array}
$$

Proof.
By Lemma .0.105 we obtain $disj_H(h_1, h_1') * (h_1, e_1) \rightarrow^* (h_1', e_1')$ for simulation in ζ'. By Lemma .0.97 we have that $h_S' \# h_1'$ thus we allocate $\mathrm{dom}(h_1') \setminus \mathrm{dom}(h_1)$ with the values specifically in h_1'. For all values in h_1 we own the points to predicate thus we can just update it directly. To update the stepping relation we use Lemma .0.95 and Lemma .0.46. □

Lemma .0.107 (Swap immutable to mutable for regions)**.**

$$
\begin{array}{l}
\forall R_1, R_2, R_3, h_1, h_2, h_3, \zeta, \zeta_1, \zeta_2, h_{3R}. \\
\quad \circledast_{r \in R_1 \uplus R_2 \uplus R_3} \left(\boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)}, h_3 = \uplus_{r \in R_3} h_{3R}(r)\right) \vdash \\
\quad heap_S(h_1 \uplus h_3, \zeta_1) * heap_S(h_2 \uplus h_3, \zeta_2) * \circledast_{i \in \{1,2\}} \circledast_{r \in R_i} [\textsc{Mu}(r, \{\zeta_i\})]^{\frac{1}{2}} * \\
\quad \circledast_{r \in R_3} (\textsc{Reg}(r) * [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h_{3R}(r))]^{\frac{3}{4}}) * \circledast_{(l,v) \in h_1 \uplus h_2 \uplus h_3} l \mapsto^{\zeta}_S v \\
\{\textsc{Rg}(r) | r \in R_1 \uplus R_2\} \Rrightarrow \{\textsc{Rg}(r) | r \in R_1 \uplus R_2 \uplus R_3\} \\
\quad heap_S(h_1 \uplus h_3, \zeta_1) * heap_S(h_2 \uplus h_3, \zeta_2) * \circledast_{r \in R_1 \uplus R_2 \uplus R_3} [\textsc{Mu}(r, \{\zeta\})]^{\frac{1}{2}}
\end{array}
$$

Lemma .0.108 (Complete Non-interference parallelization)**.**

$$
\begin{array}{ll}
& \forall \zeta, \zeta_1, \zeta_2, \Lambda_1, \Lambda_2, \Lambda_3, M, e_1, e_2, v_1, v_2, j, h_1, h_2, h_3, h_{3R}, \pi, \varepsilon_1, \varepsilon_2, R, S. \\
& \quad R = I(M(\Lambda_1 \uplus \Lambda_2 \uplus \Lambda_3)), S = \{\textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\} \vdash \\
& \qquad S(\zeta_1, 0, h_1, e_1, v_1, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{\mathbf{1}}{2}, \varepsilon_1, M) * \\
& \qquad S(\zeta_2, 0, h_2, e_2, v_2, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{\mathbf{1}}{2}, \varepsilon_2, M) * \\
& \qquad P_f(\Lambda_3, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * \\
& \qquad [\textsc{Sr}]^{\pi}_{\zeta} * \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)} \\
\Rrightarrow_{R \uplus S} & \quad j \overset{\zeta}{\Rightarrow}_S (v_1, v_2) * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * \\
& \quad P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}
\end{array}
$$

Proof.

Context: $\zeta, \zeta_1, \zeta_2, R, \Lambda_1, \Lambda_2, \Lambda_3, M, e_0, e_1, e_2, v_1, v_2, j, h_0, h_1, h_2, h_3, h_{3R}, \pi, \varepsilon_1, \varepsilon_2$

$S(\zeta_1, 0, h_1, e_1, v_1, \frac{1}{2}, (\Lambda_1, \Lambda_3), \frac{1}{2}, \varepsilon_1, M) * S(\zeta_2, 0, h_2, e_2, v_2, \frac{1}{2}, (\Lambda_2, \Lambda_3), \frac{1}{2}, \varepsilon_2, M) *$

$P_f(\Lambda_3, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * [\textsc{Sr}]^{\pi}_{\zeta} * \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}$

$\Rrightarrow_{R \uplus \{\textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$ // Unfold S and P_f

Context: $\boxed{\textsc{Spec}(h_1, e_1, \zeta_1)}^{\textsc{Sp}(\zeta_1)}, \boxed{\textsc{Spec}(h_2, e_2, \zeta_2)}^{\textsc{Sp}(\zeta_2)}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}$

$0 \overset{\zeta_1}{\Rightarrow}_S v_1 * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_1} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta_1) * P_{reg}(\Lambda_3, \frac{1}{2}, \varepsilon_1, M, \zeta_1) *$

$0 \overset{\zeta_2}{\Rightarrow}_S v_2 * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_2} * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta_2) * P_{reg}(\Lambda_3, \frac{1}{2}, \varepsilon_2, M, \zeta_2) *$

$h_3 = \uplus_{r \in M(\Lambda_3)} h_{3R}(r) * heaps(\{\zeta\}, h_1 \uplus h_2 \uplus h_3) * \circledast_{\rho \in \mathsf{rds}\ \varepsilon_1 \cup \varepsilon_2 \setminus \varepsilon_1 \cap \varepsilon_2} [\textsc{Rd}]^{\frac{1}{2}}_{M(\rho)} *$

$\circledast_{r \in M(\Lambda_3)} [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h_{3R}(r))]^{\frac{1}{4}} * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

$\Rrightarrow_{R \uplus \{\textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$ // Lemma .0.102

$0 \overset{\zeta_1}{\Rightarrow}_S v_1 * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_1} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta_1) * 0 \overset{\zeta_2}{\Rightarrow}_S v_2 * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_2} * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta_2) *$

$h_3 = \uplus_{r \in M(\Lambda_3)} h_{3R}(r) * heaps(\{\zeta\}, h_1 \uplus h_2 \uplus h_3) * \circledast_{r \in M(\Lambda_3)} [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h(r))]^{\frac{3}{4}} *$

$P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) * P_{regs}(\Lambda_3, M) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

$\Rrightarrow_R$ // Lemma .0.103

$\exists h_0, h_S, e_0, e_S, e_1, e_2, h_1', h_2', v_1, v_2.\ \textsc{Spec}(h_0, h_S \uplus h_1 \uplus h_2 \uplus h_3, e_0, e_S, \frac{1}{2}, \zeta) *$

$\textsc{Spec}(h_1 \uplus h_3, h_1', e_1, v_1, \frac{1}{2}, \zeta_1) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_1} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta_1) *$

$\textsc{Spec}(h_2 \uplus h_3, h_2', e_2, v_2, \frac{1}{2}, \zeta_2) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_2} * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta_2) *$

$h_3 = \uplus_{r \in M(\Lambda_3)} h_{3R}(r) * heaps(\{\zeta\}, h_1 \uplus h_2 \uplus h_3) * \circledast_{r \in M(\Lambda_3)} [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h_{3R}(r))]^{\frac{3}{4}} *$

$P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) * P_{regs}(\Lambda_3, M) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

$\Rrightarrow_{\{\textsc{Rg}(r) | r \in M(\Lambda_1 \uplus \Lambda_2)\}}$ // Lemma .0.104

$\exists h_0, h_S, e_0, e_S, e_1, e_2, h_1', h_2', v_1, v_2.\ \textsc{Spec}(h_0, h_S \uplus h_1 \uplus h_2 \uplus h_3, e_0, e_S, \frac{1}{2}, \zeta) *$

$\textsc{Spec}(h_1 \uplus h_3, h_1' \uplus h_3, e_1, v_1, \frac{1}{2}, \zeta_1) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_1} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta_1) *$

$\textsc{Spec}(h_2 \uplus h_3, h_2' \uplus h_3, e_2, v_2, \frac{1}{2}, \zeta_2) * [\textsc{Sr}]^{\frac{1}{2}}_{\zeta_2} * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta_2) *$

$h_3 = \uplus_{r \in M(\Lambda_3)} h_{3R}(r) * heaps(\{\zeta\}, h_1 \uplus h_2 \uplus h_3) * \circledast_{r \in M(\Lambda_3)} [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h_{3R}(r))]^{\frac{3}{4}} *$

$P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) * \circledast_{r \in M(\Lambda_3)} \textsc{Reg}(r) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

$\Rrightarrow_{\{\textsc{Rg}(r) | r \in M(\Lambda_1 \uplus \Lambda_2)\}}$ // Lemma .0.106 with $k_1 \overset{\zeta}{\Rightarrow}_S e_1$ and $k_2 \overset{\zeta}{\Rightarrow}_S e_2$

$\exists h_0, h_S, e_0, e_S', e_1, e_2, h_1', h_2', v_1, v_2.\ \textsc{Spec}(h_0, h_S \uplus h_1' \uplus h_2' \uplus h_3, e_0, e_S', \frac{1}{2}, \zeta) *$

$\textsc{Spec}(h_1 \uplus h_3, h_1' \uplus h_3, e_1, v_1, 1, \zeta_1) * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta_1) *$

$\textsc{Spec}(h_2 \uplus h_3, h_2' \uplus h_3, e_2, v_2, 1, \zeta_2) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta_2) *$

$h_3 = \uplus_{r \in M(\Lambda_3)} h_{3R}(r) * heaps(\{\zeta\}, h_1' \uplus h_2' \uplus h_3) *$

$\circledast_{r \in M(\Lambda_3)} [\textsc{Im}(r, \{\zeta_1, \zeta_2\}, h_{3R}(r))]^{\frac{3}{4}} * P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) *$

$\circledast_{r \in M(\Lambda_3)} \textsc{Reg}(r) * j \overset{\zeta}{\Rightarrow}_S (v_1, v_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

$\Rrightarrow_R$ // Lemma .0.107

$\exists h_0, h_S, e_0, e'_S, e_1, e_2, h'_1, h'_2, v_1, v_2.\ \textsc{Spec}(h_0, h_S \uplus h'_1 \uplus h'_2 \uplus h_3, e_0, e'_S, \frac{1}{2}, \zeta) *$
$\textsc{Spec}(h_1 \uplus h_3, h'_1 \uplus h_3, e_1, v_1, 1, \zeta_1) * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) *$
$\textsc{Spec}(h_2 \uplus h_3, h'_2 \uplus h_3, e_2, v_2, 1, \zeta_2) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) *$
$\circledast_{r \in M(\Lambda_3)} [\textsc{Mu}(r, \{\zeta\})]^{\frac{3}{4}} * \circledast_{r \in M(\Lambda)} \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} *$
$P_{effs}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M) * j \overset{\zeta}{\Rightarrow}_S (v_1, v_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

$\Rrightarrow_{R \uplus \{\textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$
$\boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)} * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) *$
$P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) * j \overset{\zeta}{\Rightarrow}_S (v_1, v_2) * [\textsc{Sr}]^{\pi}_{\zeta}$

□

Proof of Parallelization.

Let $\Lambda = \Lambda_1, \Lambda_2, \Lambda_3$ and we have to show $\mathcal{E}^{\cdot;\Lambda}_{\varepsilon_1 \cup \varepsilon_2, M}(\tau_1 \times \tau_2)(e_{1I} \| e_{2I}, (e_{1S}, e_{2S}))$:

Context: $j, \pi, \zeta, h_0, e_0, \boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}$

$$\left\{ j \overset{\zeta}{\Rightarrow}_S (e_{1S}, e_{2S}) * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta)\}}$$

Open $\textsc{Sp}(\zeta)$

$$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S (e_{1S}, e_{2S}) * [\textsc{Sr}]^{\pi}_{\zeta} * \triangleright(\textsc{Spec}(h_0, e_0, \zeta)) * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * \\ P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array} \right\}_{\{\textsc{Hp}\}}$$

// Lemma .0.101

$$\left\{ \begin{array}{l} \exists \zeta_1, \zeta_2, h_1, h_2, h_3, h_{3R}.\ \textsc{Spec}(h_0, e_0, \zeta) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * \\ [\textsc{Sr}]^{\pi}_{\zeta} * S(\zeta_1, 0, h_1 \uplus h_3, e_{1S}, e_{1S}, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{1}{2}, \varepsilon_1, M) * \\ S(\zeta_2, 0, h_2 \uplus h_3, e_{2S}, e_{2S}, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{1}{2}, \varepsilon_2, M) * \\ P_f(\Lambda_3, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$$

$$\left\{ \begin{array}{l} \exists \zeta_1, \zeta_2, h_1, h_2, h_3, h_{3R}. \\ S(\zeta_1, 0, h_1 \uplus h_3, e_{1S}, e_{1S}, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{1}{2}, \varepsilon_1, M) * \\ S(\zeta_2, 0, h_2 \uplus h_3, e_{2S}, e_{2S}, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{1}{2}, \varepsilon_2, M) * \\ j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * [\textsc{Sr}]^{\pi}_{\zeta} * \\ P_f(\Lambda_3, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$$

$e_1 \| e_2$

$$\left\{ S(\zeta_1, 0, h_1 \uplus h_3, e_{1S}, e_{1S}, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{1}{2}, \varepsilon_1, M) \right\}$$

e_{1I}

$$\left\{ \begin{array}{l} v_{1I}.\ \exists v_{1S}.\ S(\zeta_1, 0, h_1 \uplus h_3, e_{1S}, v_{1S}, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{1}{2}, \varepsilon_1, M) * \\ [\![\tau_1]\!]^M(v_{1I}, v_{1S}) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$$

$$\left\{ S(\zeta_2, 0, h_2 \uplus h_3, e_{2S}, e_{2S}, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{1}{2}, \varepsilon_2, M) \right\}$$

e_{2I}

$$\left\{ \begin{array}{l} v_{2I}.\ \exists v_{2S}.\ S(\zeta_2, 0, h_2 \uplus h_3, e_{2S}, v_{2S}, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{1}{2}, \varepsilon_2, M) * \\ [\![\tau_2]\!]^M(v_{2I}, v_{2S}) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$$

$$\left\{ \begin{array}{l} v.\ v = (v_{1I}, v_{2I}) * \exists v_{1S}, v_{2S}. \\ S(\zeta_1, 0, h_1 \uplus h_3, e_{1S}, v_{1S}, \tfrac{1}{2}, (\Lambda_1, \Lambda_3), \tfrac{1}{2}, \varepsilon_1, M) * \\ S(\zeta_2, 0, h_2 \uplus h_3, e_{2S}, v_{2S}, \tfrac{1}{2}, (\Lambda_2, \Lambda_3), \tfrac{1}{2}, \varepsilon_2, M) * j \overset{\zeta}{\Rightarrow}_S (e_1, e_2) * \\ [\textsc{Sr}]^{\pi}_{\zeta} * P_f(\Lambda_3, M, \zeta, \zeta_1, \zeta_2, \varepsilon_1, \varepsilon_2, h_1, h_2, h_3, h_{3R}) * \\ [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * [\![\tau_2]\!]^M(v_{2I}, v_{2S}) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Sp}(\zeta_1), \textsc{Sp}(\zeta_2)\}}$$ //

Lemma .0.108

$$\left\{ \begin{array}{l} v.\ v = (v_{1I}, v_{2I}) * \exists v_{1S}, v_{2S}.\ j \overset{\zeta}{\Rightarrow}_S (v_{1S}, v_{2S}) * [\textsc{Sr}]^{\pi}_{\zeta} * [\![\tau_1]\!]^M(v_{1I}, v_{1S}) * \\ [\![\tau_2]\!]^M(v_{2I}, v_{2S}) * P_{reg}(\Lambda_1, \mathbf{1}, \varepsilon_1, M, \zeta) * P_{reg}(\Lambda_2, \mathbf{1}, \varepsilon_2, M, \zeta) * \\ P_{reg}(\Lambda_3, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta)\}}$$

$$\left\{ \begin{array}{l} v.\ v = (v_{1I}, v_{2I}) * \exists v_{1S}, v_{2S}.\ j \overset{\zeta}{\Rightarrow}_S (v_{1S}, v_{2S}) * \\ [\![\tau_1 \times \tau_2]\!]^M((v_{1I}, v_{2I}), (v_{2I}, v_{2S})) * [\textsc{Sr}]^{\pi}_{\zeta} * P_{reg}(\Lambda, \mathbf{1}, \varepsilon_1 \cup \varepsilon_2, M, \zeta) \end{array} \right\}_{\{\textsc{Hp}, \textsc{Sp}(\zeta)\}}$$

□

Commuting

Assuming

1. $\Lambda_3 \mid \Lambda_1 \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1$
2. $\Lambda_3 \mid \Lambda_2 \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2$
3. $\mathsf{als}\ \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{als}\ \varepsilon_2 \subseteq \Lambda_2$, $\mathsf{wrs}\ \varepsilon_1 \subseteq \Lambda_1$, $\mathsf{wrs}\ \varepsilon_2 \subseteq \Lambda_2$, $\mathsf{rds}\ \varepsilon_1 \subseteq \Lambda_1 \cup \Lambda_3$ and $\mathsf{rds}\ \varepsilon_2 \subseteq \Lambda_2 \cup \Lambda_3$

then

$$\cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma \vdash (e_1, e_2) \preceq \mathsf{let}\ x = e_2\ \mathsf{in}\ (e_1, x) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$$

Proof. By parallelization, we have

$$\cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma \vdash (e_1, e_2) \preceq e_1 \parallel e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$$

and by switching the parallel composition

$$\cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma \vdash e_1 \parallel e_2 \preceq \mathsf{let}\ x = e_2 \parallel e_1\ \mathsf{in}\ (\pi_2(x), \pi_1(x)) : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2$$

now using parallel composition in the opposite direction

$$\begin{aligned} \cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma \vdash \mathsf{let}\ x = e_2 \parallel e_1\ \mathsf{in}\ (\pi_2(x), \pi_1(x)) \preceq \\ \mathsf{let}\ x = (e_2, e_1)\ \mathsf{in}\ (\pi_2(x), \pi_1(x))\ :\ \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2 \end{aligned}$$

for which the post-condition easily follows

$$\begin{aligned} \cdot \mid \Lambda_1, \Lambda_2, \Lambda_3 \mid \Gamma \vdash \mathsf{let}\ x = (e_2, e_1)\ \mathsf{in}\ (\pi_2(x), \pi_1(x)) \preceq \\ \mathsf{let}\ x = e_2\ \mathsf{in}\ (e_1, x)\ :\ \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2 \end{aligned}$$

□

Data Abstraction

Example: Stacks

Consider the following two stack-modules:

$Stack_1$ has a single reference to a pure functional list, where the **cas** operation is used to update the entire list on push and pop.

$$\begin{aligned} create_1() &= \mathsf{let}\ h = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ (push_1, pop_1) \\ push_1(n) &= \mathsf{let}\ v = !h\ \mathsf{in} \\ &\qquad \mathsf{let}\ v' = \mathsf{inj}_2\ (n, v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h, v, v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_1(n) \\ pop_1() &= \mathsf{let}\ v = !h\ \mathsf{in} \\ &\qquad \mathsf{case}(v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\ &\qquad\qquad \mathsf{inj}_2\ (n, v') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h, v, v')\ \mathsf{then}\ \mathsf{inj}_2\ n\ \mathsf{else}\ pop_1()) \end{aligned}$$

$Stack_2$ uses a header-reference to a mutable linked list, where the **cas** operation is used to move the header back on pop and forth on push.

$$
\begin{aligned}
create_2() &= \mathsf{let}\ t = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ \mathsf{let}\ h = \mathsf{new}\ t\ \mathsf{in}\ (push_2, pop_2)\\
push_2(n) &= \mathsf{let}\ v = !h\ \mathsf{in}\\
&\qquad \mathsf{let}\ v' = \mathsf{new}\ \mathsf{inj}_2\ (n, v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h, v, v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_2(n)\\
pop_2() &= \mathsf{let}\ v = !h\ \mathsf{in}\\
&\qquad \mathsf{let}\ v' = !v\ \mathsf{in}\\
&\qquad\quad \mathsf{case}(v', \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (),\\
&\qquad\qquad\quad \mathsf{inj}_2\ (n, v'') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h, v, v'')\ \mathsf{then}\ \mathsf{inj}_2\ n\ \mathsf{else}\ pop_2())
\end{aligned}
$$

The physical footprint of the two modules differ, thus to show contextual equivalence we are required to establish an invariant that relates one location having a pure functional list to a collection of mutable heap-cells organized as a linked list. Such equivalences was not possible to show in 'A Concurrent Logical Relation' due to their more restrictive worlds allowing invariants to only relate values at two locations for a semantic type.

Theorem .0.109 ($Stack_1$ and $Stack_2$ are contextually equivalent)**.**

$$\forall \tau.\ \rho \mid \cdot \mid \cdot \vdash create_1 \cong_{ctx} create_2 : \mathbf{1} \to^{\rho \mid \cdot}_{al_\rho} (\tau \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau), \emptyset$$

Proof. Contextual equivalence is defined as contextual approximation in both directions, thus we are to show:

$$\rho \mid \cdot \mid \cdot \vdash create_1 \leq_{ctx} create_2 : \mathbf{1} \to^{\rho \mid \cdot}_{al_\rho} (\tau \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau), \emptyset \quad (5)$$

$$\rho \mid \cdot \mid \cdot \vdash create_2 \leq_{ctx} create_1 : \mathbf{1} \to^{\rho \mid \cdot}_{al_\rho} (\tau \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau), \emptyset \quad (6)$$

(1) follows from Lemma .0.110 and soundness. Similarly, (2) follows from Lemma .0.111 and soundness.

□

Lemma .0.110 ($Stack_1$ logically refines $Stack_2$)**.**

$$\forall \tau.\ \rho \mid \cdot \mid \cdot \vdash create_1 \preceq create_2 : \mathbf{1} \to^{\rho \mid \cdot}_{al_\rho} (\tau \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau), \emptyset$$

Proof. Proof follows directly from Lemma .0.113 and Lemma .0.114 □

Lemma .0.111 ($Stack_2$ logically refines $Stack_1$)**.**

$$\forall \tau.\ \rho \mid \cdot \mid \cdot \vdash create_2 \preceq create_1 : \mathbf{1} \to^{\rho \mid \cdot}_{al_\rho} (\tau \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho \mid \cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau), \emptyset$$

Proof. This direction is straight-forward, since any successful update from **cas** forces the shape of the linked list on the implementation side and we are required to make only a single heap update on the specification side for $Stack_1$: □

We choose the following relation to show equality:

$$\textsc{StackRel}(h,r,\phi) \triangleq \exists l, v, n.\ h_I \overset{1}{\hookrightarrow}_{I,r} v * h_S \overset{1}{\hookrightarrow}_{S,r} n * vals(l,v,\phi) * linked(l,n,r,\phi)$$

$$\textsc{StackInv}(h,r,\phi) \triangleq \exists \iota.\ \boxed{\textsc{StackRel}(h,r,\phi)}^{\textsc{Si}(\iota)}$$

where

$$vals(\mathsf{nil},v,\phi) \triangleq v = \mathbf{inj}_1\ ()$$

$$vals(x :: xs,v,\phi) \triangleq \exists v'.\ v = \mathbf{inj}_2\ (x_I, v') * \phi(x) * vals(xs,v',\phi)$$

and

$$linked(\mathsf{nil},n,r,\phi) \triangleq \exists v.\ n \overset{1}{\hookrightarrow}_{S,r} v * v = \mathbf{inj}_1\ ()$$

$$linked(x :: xs,n,r,\phi) \triangleq \exists v, n'.\ n \overset{1}{\hookrightarrow}_{S,r} v * v = \mathbf{inj}_2\ (x_S, n') * \phi(x) * linked(xs,n',r,\phi)$$

and the function $\textsc{Si}(\iota)$ ensures that the invariant identifier is disjoint from $\textsc{Hp}$,$\textsc{Sp}(\zeta)$ and $\textsc{Rg}(r)$ for all ζ and r.

Lemma .0.112 (Can create StackInv)**.**

$$\begin{array}{ll} & \forall h_I, h_S, l_S, r, \phi. \\ & \quad h_I \overset{1}{\hookrightarrow}_{I,r} \mathit{inj}_1\ () * l_S \overset{1}{\hookrightarrow}_{S,r} \mathit{inj}_1\ () * h_S \overset{1}{\hookrightarrow}_{S,r} l_S \\ \Rrightarrow^{\textsc{Si}(\iota)} & \textsc{StackInv}((h_I,h_S),r,\phi) \end{array}$$

Proof. Intro h_I, h_S, l_S, r and ϕ.

$$\begin{array}{ll} & h_I \overset{1}{\hookrightarrow}_{I,r} \mathbf{inj}_1\ () * l_S \overset{1}{\hookrightarrow}_{S,r} \mathbf{inj}_1\ () * h_S \overset{1}{\hookrightarrow}_{S,r} l_S \\ \Rrightarrow & \exists v_I.\ v_I = \mathbf{inj}_1\ () * h_I \overset{1}{\hookrightarrow}_{I,r} v_I * l_S \overset{1}{\hookrightarrow}_{S,r} \mathbf{inj}_1\ () * h_S \overset{1}{\hookrightarrow}_{S,r} l_S * vals(\mathsf{nil}, v_I, \phi) \\ \Rrightarrow & \exists v_I, v_S.\ h_I \overset{1}{\hookrightarrow}_{I,r} v_I * h_S \overset{1}{\hookrightarrow}_{S,r} v_S * vals(\mathsf{nil}, v_I, \phi) * linked(\mathsf{nil}, v_S, \phi) \\ \Rrightarrow & \textsc{StackRel}((h_I,h_S),r,\phi) \\ \Rrightarrow^{\textsc{Si}(\iota)} & \exists \iota.\ \boxed{\textsc{StackRel}((h_I,h_S),r,\phi)}^{\textsc{Si}(\iota)} \\ \Rrightarrow & \textsc{StackInv}((h_I,h_S),r,\phi) \end{array}$$

□

Lemma .0.113. *$Stack_1$-push refines $Stack_2$-push*

$$\begin{array}{ll} & \forall \rho, M, h, n, m.\ V[\![\tau]\!]^M(n,m) \\ \Rightarrow & \textsc{StackInv}(h, M(\rho)) \vdash E^{\rho;\cdot}_{\{al_\rho, wr_\rho, rd_\rho\};M}(V[\![\mathbf{1}]\!]^M)(push_1(n), push_2(m)) \end{array}$$

Proof. We define the following short-hands:

$e_{1I} \triangleq \mathsf{let}\ v = !h_I\ \mathsf{in}\ \mathsf{let}\ v' = \mathsf{inj}_2\ (n,v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h,v,v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_1(n)$

$e_{1S} \triangleq \mathsf{let}\ v = !h_I\ \mathsf{in}\ \mathsf{let}\ v' = \mathsf{new}\ \mathsf{inj}_2\ (m,v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h,v,v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_2(m)$

$K_{1I} \triangleq \mathsf{let}\ v = []\ \mathsf{in}\ \mathsf{let}\ v' = \mathsf{inj}_2\ (n,v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h,v,v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_1(n)$

$K_{2I} \triangleq \mathsf{let}\ v' = []\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h,v_I^1,v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_1(n)$

$K_{3I} \triangleq \mathsf{if}\ []\ \mathsf{then}\ ()\ \mathsf{else}\ push_1(n)$

$K_{1S} \triangleq \mathsf{let}\ v = []\ \mathsf{in}\ \mathsf{let}\ v' = \mathsf{new}\ \mathsf{inj}_2\ (n,v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h,v,v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_2(m)$

$K_{2S} \triangleq \mathsf{let}\ v' = []\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h,v_S^1,v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_2(m)$

$K_{3S} \triangleq \mathsf{if}\ []\ \mathsf{then}\ ()\ \mathsf{else}\ push_2(m)$

and the following predicate to track the stacks:

$$\textsc{StackRel}(h,l,l',v,n,r,\phi) \triangleq h_I \overset{1}{\hookrightarrow}_{I,r} v * h_S \overset{1}{\hookrightarrow}_{S,r} n * vals(l,v,\phi) * linked(l',n,r,\phi)$$

and continue by Löb-induction.

Context: $g, j, \pi', e_0, h_0, \zeta, M, h, n, m$

Context: $\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0,e_0,\zeta)}^{\textsc{Sp}(\zeta)}, \textsc{StackInv}(h, M(\rho), V[\![\tau]\!]^M), V[\![\tau]\!]^M(n,m),$
$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) \right\}$

▷ $push(n)$

$\left\{ v_I^1.\ \exists v_S^1.\ j \overset{\zeta}{\Rightarrow}_S v_S^1 * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) * V[\![\mathbf{1}]\!]^M(v_I^1, v_S^1) \right\}_{\top}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) \right\}_{\top}$

// Let $\pi = g(\rho)$, $r = M(\rho)$ and $R = \{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rg}(r)\}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} \right\}_{\top}$

Bind on $K_{1I}[!h_I]$

// Unfolding $\textsc{StackInv}(h, r, V[\![\tau]\!]^M)$

$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \exists \iota.\ \boxed{\textsc{StackRel}(h,r,V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)} \end{array} \right\}_{\top}$

Open $R, \textsc{Si}(\iota)$

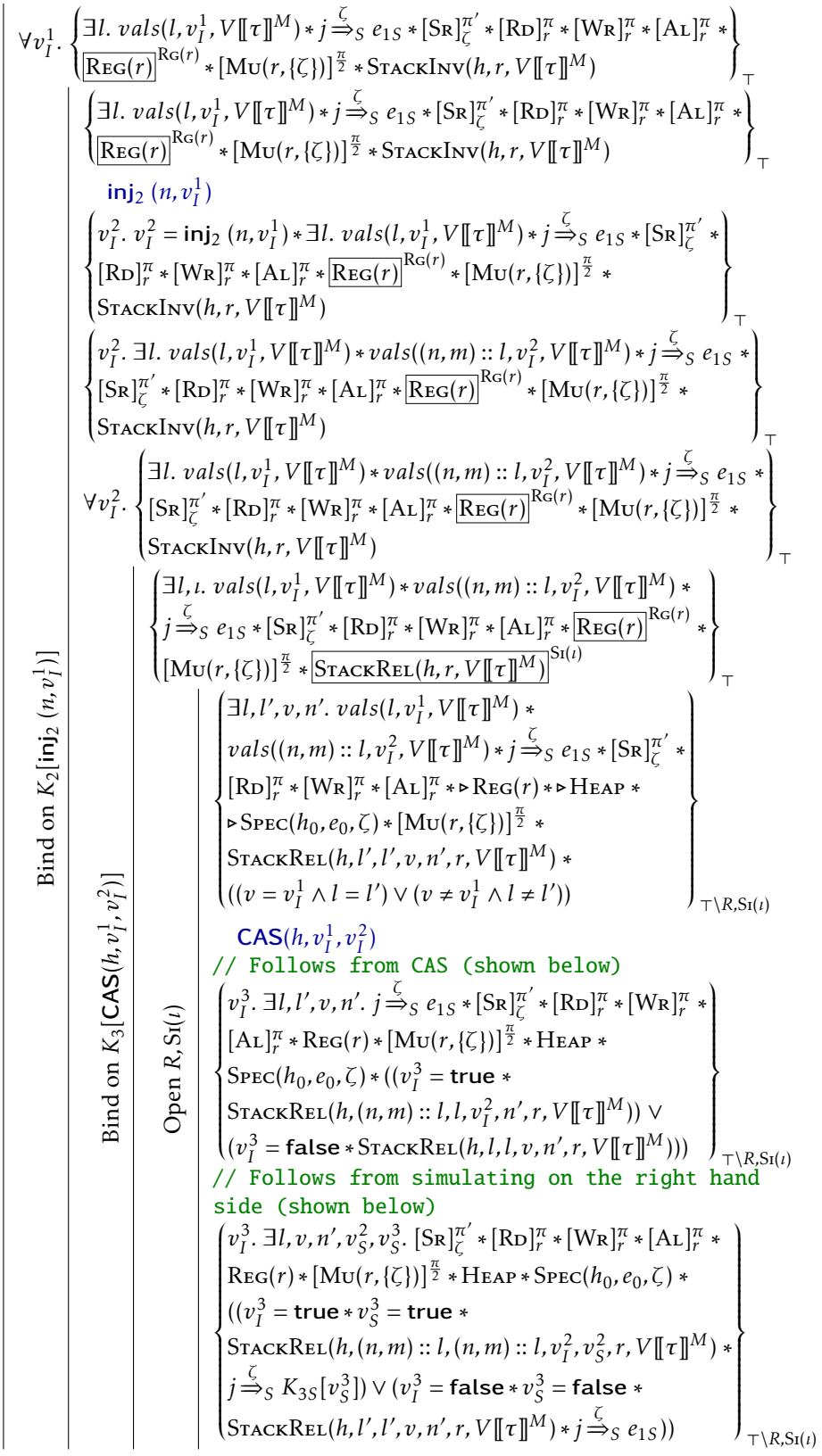
$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * [\textsc{Al}]^{\pi}_{r} * \triangleright \textsc{Reg}(r) * \\ [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}(h_0,e_0,\zeta) * \\ \triangleright \textsc{StackRel}(h,r,V[\![\tau]\!]^M) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$!h_I$

// Follows from Lemma .0.63

$\left\{ \begin{array}{l} v_I^1.\ \exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * \\ [\textsc{Wr}]^{\pi}_{r} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \\ \textsc{Spec}(h_0,e_0,\zeta) * \textsc{StackRel}(h,r,V[\![\tau]\!]^M) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$\left\{ \begin{array}{l} v_I^1.\ \exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * [\textsc{Al}]^{\pi}_{r} * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h,r,V[\![\tau]\!]^M) \end{array} \right\}_{\top}$


$\forall v_I^1.$ $\left\{\exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M)\right\}_\top$
Bind on $K_2[\mathsf{inj}_2\ (n, v_I^1)]$
$\mathsf{inj}_2\ (n, v_I^1)$
$\left\{v_I^2.\ v_I^2 = \mathsf{inj}_2\ (n, v_I^1) * \exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M)\right\}_\top$
$\left\{v_I^2.\ \exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M)\right\}_\top$
$\forall v_I^2.$ $\left\{\exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M)\right\}_\top$
Bind on $K_3[\mathsf{CAS}(h, v_I^1, v_I^2)]$
$\left\{\exists l, \iota.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \boxed{\textsc{StackRel}(h, r, V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)}\right\}_\top$
Open $R, \textsc{Si}(\iota)$
$\left\{\exists l, l', v, n'.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \triangleright \textsc{Reg}(r) * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackRel}(h, l', l', v, n', r, V[\![\tau]\!]^M) * ((v = v_I^1 \wedge l = l') \vee (v \neq v_I^1 \wedge l \neq l'))\right\}_{\top \setminus R, \textsc{Si}(\iota)}$
$\mathsf{CAS}(h, v_I^1, v_I^2)$
// Follows from CAS (shown below)
$\left\{v_I^3.\ \exists l, l', v, n'.\ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * ((v_I^3 = \mathsf{true} * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, n', r, V[\![\tau]\!]^M)) \vee (v_I^3 = \mathsf{false} * \textsc{StackRel}(h, l, l, v, n', r, V[\![\tau]\!]^M)))\right\}_{\top \setminus R, \textsc{Si}(\iota)}$
// Follows from simulating on the right hand side (shown below)
$\left\{v_I^3.\ \exists l, v, n', v_S^2, v_S^3.\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * ((v_I^3 = \mathsf{true} * v_S^3 = \mathsf{true} * \textsc{StackRel}(h, (n, m) :: l, (n, m) :: l, v_I^2, v_S^2, r, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S K_{3S}[v_S^3]) \vee (v_I^3 = \mathsf{false} * v_S^3 = \mathsf{false} * \textsc{StackRel}(h, l', l', v, n', r, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S}))\right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$$\left\{\begin{array}{l} v_I^3.\ \exists v_S^3.\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * ((v_I^3 = \mathsf{true} * \\ v_S^3 = \mathsf{false} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[v_S^3]) \vee \\ (v_I^3 \neq \mathsf{true} * v_S^3 = \mathsf{false} * j \overset{\zeta}{\Rightarrow}_S e_{1S})) \end{array}\right\}_\top$$

$\mathsf{if}\ v_I^3\ \mathsf{then}$

$$\left\{\begin{array}{l} [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[()] \end{array}\right\}_\top$$

$()$

$$\left\{\begin{array}{l} v_I^4.\ \exists v_S^3.\ j \overset{\zeta}{\Rightarrow}_S v_S^3 * [\textsc{Sr}]_\zeta^{\pi'} * V[\![\mathbf{1}]\!]^M(v_I^4, v_S^3) * \\ P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) \end{array}\right\}_\top$$

else

$$\left\{j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta)\right\}_\top$$

$push(n)$

// Follows from IH

$$\left\{\begin{array}{l} v_I^4.\ \exists v_S^3.\ j \overset{\zeta}{\Rightarrow}_S v_S^3 * [\textsc{Sr}]_\zeta^{\pi'} * V[\![\mathbf{1}]\!]^M(v_I^4, v_S^3) * \\ P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) \end{array}\right\}_\top$$

We have to show we can perform the **cas** (open invariants are $R, \textsc{Si}(\iota)$):

$$\left\{\begin{array}{l} \exists l, l', v, n'.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * \\ [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \triangleright \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}(h_0, e_0, \zeta) * \\ \textsc{StackRel}(h, l', l', v, n', r, V[\![\tau]\!]^M) * ((v = v_I^1 \wedge l = l') \vee (v \neq v_I^1 \wedge l \neq l')) \end{array}\right\}$$

$$\left\{\begin{array}{l} \exists l, l', v, n'.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * \\ [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * \\ \textsc{StackRel}(h, l', l', v, n', r, V[\![\tau]\!]^M) * ((v = v_I^1 \wedge l = l') \vee (v \neq v_I^1 \wedge l \neq l')) \end{array}\right\}$$

$$\left\{\begin{array}{l} \exists l, l', v, n'.\ vals(l, v_I^1, V[\![\tau]\!]^M) * vals((n, m) :: l, v_I^2, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * \\ [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \exists t.\ locs((t_I[h_I \mapsto v], t_S[h_S \mapsto n']), r) * toks(1, 1, r) * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackRel}(h, l, l, v, n', r, V[\![\tau]\!]^M) * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * \\ ((v = v_I^1 \wedge l = l') \vee (v \neq v_I^1 \wedge l \neq l')) \end{array}\right\}$$

Frame

$$\left\{\textsc{Heap} * h_I \mapsto_I v\right\}$$

$\mathsf{CAS}(h_I, v_I^1, v_I^2)$

$$\left\{v_I^3.\ \textsc{Heap} * ((v_I^3 = \mathsf{true} * h_I \mapsto_I v_I^2) \vee (v_I^3 = \mathsf{false} * h_I \mapsto_I v))\right\}$$

// Updating $h \overset{1}{\hookrightarrow}_{I,r} v$ follows from having the authorative element and fragment

$$\left\{\begin{array}{l} v_I^3.\ \exists l, l', v, n'.\ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * [\textsc{Al}]_r^\pi * \textsc{Heap} * \\ \textsc{Spec}(h_0, e_0, \zeta) * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\ ((v_I^3 = \mathsf{true} * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, n', r, V[\![\tau]\!]^M)) \vee \\ (v_I^3 = \mathsf{false} * \textsc{StackRel}(h, l', l', v, n', r, V[\![\tau]\!]^M))) \end{array}\right\}$$

We also have to show that we could simulate on the right hand side, which consists of three parts - (1) reading the head pointer, (2) allocating a new location for the new node and (3) updating the head pointer:

$$
\begin{array}{ll}
 & \exists l, n'.\ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Spec}(h_0, e_0, \zeta) * \\
 & \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, n', r, V[\![\tau]\!]^M) \\
\Rrightarrow & \exists l, n'.\ j \overset{\zeta}{\Rightarrow}_S K_{1S}[!h_S] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\
 & \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, n', r, V[\![\tau]\!]^M) \\
 & \texttt{// Follows from Lemma .0.65} \\
\Rrightarrow & \exists l, n', v_S^1.\ j \overset{\zeta}{\Rightarrow}_S K_{1S}[v_S^1] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\
 & \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, v_S^1, r, V[\![\tau]\!]^M) \\
\Rrightarrow & \exists l, n', v_S^1.\ j \overset{\zeta}{\Rightarrow}_S K_{2S}[\mathsf{new}\ \mathsf{inj}_2\ (n, v_S^1)] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * \\
 & [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, v_S^1, r, V[\![\tau]\!]^M) \\
 & \texttt{// Follows from Lemma .0.79} \\
\Rrightarrow & \exists l, n', v_S^1, v_S^2.\ j \overset{\zeta}{\Rightarrow}_S K_{2S}[v_S^2] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\
 & \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, v_S^1, r, V[\![\tau]\!]^M) * \\
 & v_S^2 \mapsto^{\zeta}_S \mathsf{inj}_2\ (n, v_S^1) \\
 & \texttt{// Follows from Lemma .0.80} \\
\Rrightarrow & \exists l, n', v_S^1, v_S^2.\ j \overset{\zeta}{\Rightarrow}_S K_{2S}[v_S^2] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\
 & \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, v_S^1, r, V[\![\tau]\!]^M) * \\
 & v_S^2 \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_2\ (n, v_S^1) \\
\Rrightarrow & \exists l, n', v_S^1, v_S^2.\ j \overset{\zeta}{\Rightarrow}_S K_{3S}[\mathsf{CAS}(h_S, v_S^1, v_S^2)] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * \\
 & \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackRel}(h, (n, m) :: l, l, v_I^2, v_S^1, r, V[\![\tau]\!]^M) * \\
 & v_S^2 \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_2\ (n, v_S^1) \\
 & \texttt{// Follows from Lemma .0.62, Lemma .0.59} \\
\Rrightarrow & \exists l, n', v_S^1, v_S^2, v_S^3.\ v_S^3 = \mathsf{true} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[v_S^3] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \textsc{Reg}(r) * \\
 & \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^2 * \\
 & vals((n, m) :: l, v_I^2, \phi, V[\![\tau]\!]^M) * linked(l, v_S^1, r, V[\![\tau]\!]^M) * v_S^2 \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_2\ (n, v_S^1) \\
 & \texttt{// From } V[\![\tau]\!]^M(n, m)
\end{array}
$$

$$\begin{aligned}
\Rrightarrow\ & \exists l, n', v_S^1, v_S^2, v_S^3.\ v_S^3 = \mathsf{true} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[v_S^3] * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Al}]_r^{\pi} * \textsc{Reg}(r) * \\
& \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^2 * \\
& vals((n,m) :: l, v_I^2, \phi, V[\![\tau]\!]^M) * linked((n,m) :: l, v_S^2, r, V[\![\tau]\!]^M) \\
\Rrightarrow\ & \exists l, n', v_S^1, v_S^2, v_S^3.\ v_S^3 = \mathsf{true} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[v_S^3] * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Al}]_r^{\pi} * \textsc{Spec}(h_0, e_0, \zeta) * \\
& \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackRel}(h, (n,m) :: l, (n,m) :: l, v_I^2, v_S^2, r, V[\![\tau]\!]^M)
\end{aligned}$$

□

Lemma .0.114. *$Stack_1$-pop refines $Stack_2$-pop*

$$\begin{aligned}
& \forall \rho, M, h. \\
\Rightarrow\ & \textsc{StackInv}(h, r, V[\![\tau]\!]^M) \vdash E^{\rho;\cdot}_{wr_\rho, rd_\rho; M}(V[\![\mathbf{1} + \tau]\!]^M)(pop_1(), pop_2())
\end{aligned}$$

Proof. We define the following short-hands:

$$\begin{aligned}
e_{1I} \triangleq\ & \mathsf{let}\ v = !h_I\ \mathsf{in} \\
& \quad \mathsf{case}(v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\
& \qquad \mathsf{inj}_2\ (n_I, v') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h_I, v, v')\ \mathsf{then}\ \mathsf{inj}_2\ n_I\ \mathsf{else}\ pop_1()) \\
e_{1S} \triangleq\ & \mathsf{let}\ v = !h_S\ \mathsf{in} \\
& \quad \mathsf{let}\ v' = !v\ \mathsf{in} \\
& \quad\ \mathsf{case}(v', \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\
& \qquad \mathsf{inj}_2\ (n_S, v'') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h_S, v, v'')\ \mathsf{then}\ \mathsf{inj}_2\ n_S\ \mathsf{else}\ pop_2()) \\
K_{1I} \triangleq\ & \mathsf{let}\ v = []\ \mathsf{in}\ \mathsf{case}(v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ ()), \\
& \qquad\qquad \mathsf{inj}_2\ (n_I, v') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h_I, v, v')\ \mathsf{then}\ \mathsf{inj}_2\ n_I\ \mathsf{else}\ pop_1() \\
K_{2I} \triangleq\ & \mathsf{if}\ []\ \mathsf{then}\ \mathsf{inj}_2\ n_I\ \mathsf{else}\ pop_1() \\
K_{1S} \triangleq\ & \mathsf{let}\ v = []\ \mathsf{in} \\
& \quad \mathsf{let}\ v' = !v\ \mathsf{in} \\
& \quad\ \mathsf{case}(v', \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\
& \qquad \mathsf{inj}_2\ (n_S, v'') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h_S, v, v'')\ \mathsf{then}\ \mathsf{inj}_2\ n_S\ \mathsf{else}\ pop_2()) \\
K_{2S} \triangleq\ & \mathsf{let}\ v' = []\ \mathsf{in} \\
& \quad \mathsf{case}(v', \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\
& \qquad \mathsf{inj}_2\ (n_S, v'') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h_S, v_S^1, v'')\ \mathsf{then}\ \mathsf{inj}_2\ n_S\ \mathsf{else}\ pop_2()) \\
K_{3S} \triangleq\ & \mathsf{if}\ []\ \mathsf{then}\ \mathsf{inj}_2\ n_S\ \mathsf{else}\ pop_2()
\end{aligned}$$

Context: $g, j, \pi', e_0, h_0, \zeta, M, h$

Context: $\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \textsc{StackInv}(h, r, V[\![\tau]\!]^M)$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) \right\}$

▷ $pop()$

$\left\{ v^1_I.\ \exists v^1_S.\ j \overset{\zeta}{\Rightarrow}_S v^1_S * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) * V[\![\mathbf{1} + \tau]\!]^M(v^1_I, v^1_S) \right\}_{\top}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) \right\}_{\top}$

// Let $\pi = g(\rho)$, $r = M(\rho)$ and $R = \{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rg}(r)\}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} \right\}_{\top}$

Bind on $K_{1I}[!h_I]$

// Unfolding $\textsc{StackInv}(h, r, V[\![\tau]\!]^M)$

$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \exists \iota.\ \boxed{\textsc{StackRel}(h, r, V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)} \end{array} \right\}_{\top}$

Open $R, \textsc{Si}(\iota)$

$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * \triangleright \textsc{Reg}(r) * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}(h_0, e_0, \zeta) * \\ \triangleright \textsc{StackRel}(h, r, V[\![\tau]\!]^M) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$!h_I$

// Follows from Lemma .0.63

$\left\{ \begin{array}{l} v^1_I.\ \exists l.\ vals(l, v^1_I, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * \\ [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \\ \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, r, V[\![\tau]\!]^M) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

// Follows from simulation on the right hand side

$\left\{ \begin{array}{l} v^1_I.\ \exists l.\ vals(l, v^1_I, V[\![\tau]\!]^M) * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * \\ \textsc{Reg}(r) * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\ \textsc{StackRel}(h, r, V[\![\tau]\!]^M) * ((v^1_I = \mathsf{inj}_1\ () * j \overset{\zeta}{\Rightarrow}_S \mathsf{inj}_1\ ()) \vee \\ (\exists n_I, n_S, v^2_I.\ v^1_I = \mathsf{inj}_2\ (n_I, v^2_I) * l = (n, m) :: l' * j \overset{\zeta}{\Rightarrow}_S e_{1S})) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$\left\{ \begin{array}{l} v^1_I.\ \exists l.\ vals(l, v^1_I, V[\![\tau]\!]^M) * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^{\pi}_{r} * [\textsc{Wr}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * \\ ((v^1_I = \mathsf{inj}_1\ () * j \overset{\zeta}{\Rightarrow}_S \mathsf{inj}_1\ ()) \vee (\exists n_I, n_S, v^2_I. \\ v^1_I = \mathsf{inj}_2\ (n_I, v^2_I) * l = (n, m) :: l' * j \overset{\zeta}{\Rightarrow}_S e_{1S})) \end{array} \right\}_{\top}$

$$\forall v_I^1.\ \left\{\begin{array}{l}\exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * ((v_I^1 = \mathsf{inj}_1\ () * \\ j \overset{\zeta}{\Rightarrow}_S \mathsf{inj}_1\ ()) \vee (\exists n_I, v_I^2.\ v_I^1 = \mathsf{inj}_2\ (n_I, v_I^2) * l = (n, m) :: l' * \\ j \overset{\zeta}{\Rightarrow}_S e_{1S}))\end{array}\right\}_\top$$

case v_I^1

$\Rightarrow$ $\mathsf{inj}_1\ ()$

$$\left\{\begin{array}{l}\exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * \\ v_I^1 = \mathsf{inj}_1\ () * j \overset{\zeta}{\Rightarrow}_S \mathsf{inj}_1\ ()\end{array}\right\}_\top$$

$$\left\{\begin{array}{l}v_I^3.\ v_I^3 = \mathsf{inj}_1\ () * \exists l.\ vals(l, v_I^1, V[\![\tau]\!]^M) * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * \\ v_I^1 = \mathsf{inj}_1\ () * j \overset{\zeta}{\Rightarrow}_S \mathsf{inj}_1\ ()\end{array}\right\}_\top$$

$$\left\{\begin{array}{l}v_I^3.\ \exists v_S^3.\ j \overset{\zeta}{\Rightarrow}_S v_S^3 * [\textsc{Sr}]_\zeta^{\pi'} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) * \\ V[\![\mathbf{1} + \tau]\!]^M(v_I^3, v_S^3)\end{array}\right\}_\top$$

$\Rightarrow$ $\mathsf{inj}_2\ (n_I, v_I^2)$

$$\left\{\begin{array}{l}\exists l, l', n_S.\ vals(l, v_I^1, V[\![\tau]\!]^M) * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * \\ v_I^1 = \mathsf{inj}_2\ (n_I, v_I^2) * l = (n, m) :: l' * j \overset{\zeta}{\Rightarrow}_S e_{1S}\end{array}\right\}_\top$$

Bind on $K_{2I}[\mathsf{CAS}(h_I, v_I^1, v_I^2)]$

$$\left\{\begin{array}{l}\exists l, l', n_S, \iota.\ vals(l, v_I^1, V[\![\tau]\!]^M) * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \boxed{\textsc{StackRel}(h, r, V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)} * \\ v_I^1 = \mathsf{inj}_2\ (n_I, v_I^2) * l = (n, m) :: l' * j \overset{\zeta}{\Rightarrow}_S e_{1S}\end{array}\right\}_\top$$

Open $R, \textsc{Si}(\iota)$

$$\left\{\begin{array}{l}\exists l, l', n_S.\ vals(l, v_I^1, V[\![\tau]\!]^M) * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * \\ [\textsc{Wr}]_r^\pi * \triangleright \textsc{Heap} * \triangleright \textsc{Spec} * \triangleright \textsc{Reg}(\textsc{Rg}(r)) * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\ j \overset{\zeta}{\Rightarrow}_S e_{1S} * \triangleright \textsc{StackRel}(h, r, V[\![\tau]\!]^M) * \\ v_I^1 = \mathsf{inj}_2\ (n_I, v_I^2) * l = (n, m) :: l'\end{array}\right\}_{\top \setminus R, \textsc{Si}(\iota)}$$

$\mathsf{CAS}(h_I, v_I^1, v_I^2)$

// Follows from performing cas

$$\left\{\begin{array}{l}v_I^3.\ \exists l, l', n_S.\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \textsc{Reg}(\textsc{Rg}(r)) * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * l = (n, m) :: l' * \textsc{Heap} * \textsc{Spec} * \\ j \overset{\zeta}{\Rightarrow}_S e_{1S} * ((v_I^3 = \mathsf{true} * \\ \exists n'.\ \textsc{StackRel}(h, l', l, v_I^2, n', r, V[\![\tau]\!]^M)) \vee \\ (v_I^3 = \mathsf{false} * \textsc{StackRel}(h, r, V[\![\tau]\!]^M)))\end{array}\right\}_{\top \setminus R, \textsc{Si}(\iota)}$$

// Follows from simulating on the right hand side (below)

$$\left\{\begin{array}{l}v_I^3.\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^\pi * [\textsc{Wr}]_r^\pi * \textsc{Reg}(\textsc{Rg}(r)) * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec} * ((v_I^3 = \mathsf{true} * \\ \textsc{StackRel}(h, r, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S K_{3S}[\mathsf{true}] * \\ V[\![\tau]\!]^M(n, m)) \vee (v_I^3 = \mathsf{false} * \\ \textsc{StackRel}(h, r, V[\![\tau]\!]^M * j \overset{\zeta}{\Rightarrow}_S e_{1S})))\end{array}\right\}_{\top \setminus R, \textsc{Si}(\iota)}$$

$$\left\{\begin{array}{l} v_I^3.\ [\textsc{Sr}]_\zeta^{\pi'} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) * \\ \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * ((v_I^3 = \mathsf{true} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[\mathsf{true}] * \\ V[\![\tau]\!]^M(n,m)) \vee (v_I^3 = \mathsf{false} * j \overset{\zeta}{\Rightarrow}_S e_{1S})) \end{array}\right\}_\top$$

if v_I^3 **then**

$$\left\{\begin{array}{l} [\textsc{Sr}]_\zeta^{\pi'} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) * \\ \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S K_{3S}[\mathsf{true}] * V[\![\tau]\!]^M(n,m) \end{array}\right\}_\top$$

$\mathsf{inj}_2\ n_I$

$$\left\{\begin{array}{l} v_I^4.\ \exists v_S^4.\ j \overset{\zeta}{\Rightarrow}_S v_S^4 * [\textsc{Sr}]_\zeta^{\pi'} * V[\![\mathbf{1}+\tau]\!]^M(v_I^4, v_S^4) * \\ P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) \end{array}\right\}_\top$$

else

$$\left\{\begin{array}{l} [\textsc{Sr}]_\zeta^{\pi'} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) * \\ \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} \end{array}\right\}_\top$$

$push(h, n)$

`// Follows from IH`

$$\left\{\begin{array}{l} v_I^4.\ \exists v_S^4.\ j \overset{\zeta}{\Rightarrow}_S v_S^4 * [\textsc{Sr}]_\zeta^{\pi'} * V[\![\mathbf{1}+\tau]\!]^M(v_I^4, v_S^4) * \\ P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho, al_\rho\}, M, \zeta) \end{array}\right\}_\top$$

We have to show we can perform the simulation on the right hand side:

$$\exists l, l', n_S, n'.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$l = (n, m) :: l' * \textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S e_{1S} * \textsc{StackRel}(h, l', l, v_I^2, n', r, V[\![\tau]\!]^M)$$
$$\Rrightarrow \exists l, l', n_S, n'.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$l = (n, m) :: l' * \textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S K_{1S}[!h_S] * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{I,r} n' *$$
$$vals(l', v_I^2, V[\![\tau]\!]^M) * linked(l, n', r, V[\![\tau]\!]^M)$$
// Follows from Lemma .0.64
$$\Rrightarrow \exists l, l', n_S, v_S^1.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$l = (n, m) :: l' * \textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S K_{1S}[v_S^1] * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{I,r} v_S^1 *$$
$$vals(l', v_I^2, V[\![\tau]\!]^M) * linked(l, n', r, V[\![\tau]\!]^M)$$
// Unfolding *linked*
$$\Rrightarrow \exists l, l', n_S, v_S^1, v_S^2, n''.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$l = (n, m) :: l' * \textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S K_{2S}[!v_S^1] * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{I,r} v_S^1 *$$
$$vals(l', v_I^2, V[\![\tau]\!]^M) * linked(l', n'', r, V[\![\tau]\!]^M) * v_S^1 \overset{1}{\hookrightarrow}_{I,r} v_S^2 *$$
$$v_S^2 = \mathsf{inj}_2\ (n_S, n'') * V[\![\tau]\!]^M(n, m)$$
// Follows from Lemma .0.62, Lemma .0.59
$$\Rrightarrow \exists l, l', n_S, v_S^1, v_S^2, n''.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$l = (n, m) :: l' * \textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[\mathsf{CAS}(h_S, v_S^1, v_S^2)] * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 *$$
$$h_S \overset{1}{\hookrightarrow}_{I,r} v_S^1 * vals(l', v_I^2, V[\![\tau]\!]^M) * linked(l', n'', r, V[\![\tau]\!]^M) *$$
$$v_S^1 \overset{1}{\hookrightarrow}_{I,r} v_S^2 * v_S^2 = \mathsf{inj}_2\ (n_S, n'') * V[\![\tau]\!]^M(n, m)$$
// Perform CAS
$$\Rrightarrow \exists l, l', n_S, v_S^1, v_S^2, n''.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$l = (n, m) :: l' * \textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S K_{3S}[\mathsf{true}] * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{I,r} v_S^2 *$$
$$vals(l', v_I^2, V[\![\tau]\!]^M) * linked(l', n'', r, V[\![\tau]\!]^M) * v_S^1 \overset{1}{\hookrightarrow}_{I,r} v_S^2 *$$
$$v_S^2 = \mathsf{inj}_2\ (n_S, n'') * V[\![\tau]\!]^M(n_I, n_S, n'')$$
// Fold *linked*
$$\Rrightarrow \exists n_S, v_S^1, v_S^2.\ [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Wr}]^{\pi}_{r} * \textsc{Reg}(\textsc{Rg}(r)) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} *$$
$$\textsc{Spec} * j \overset{\zeta}{\Rightarrow}_S \mathsf{inj}_2\ n_S * \textsc{StackRel}(h, r, V[\![\tau]\!]^M) * V[\![\tau]\!]^M(n, m)$$

□

Lemma .0.115. *Create*

$$\forall \rho, M.\ E^{\rho;\cdot}_{al_\rho;M}(V[\![\tau \to^{\rho|\cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho|\cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau]\!]^M)(create_1(), create_2())$$

Proof.

Context: g, j, K, π', ζ, M

Context: $\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S \mathsf{let}\ t = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ \mathsf{let}\ h = \mathsf{new}\ t\ \mathsf{in}\ (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ P_{reg}(\{\rho\}, g, \{al_\rho\}, M, \zeta) \end{array}\right\}_{\top}$$

// Let $r = M(\rho)$ and $R = \{\textsc{Hp}, \textsc{Sp}(\zeta), r\}$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S \mathsf{let}\ t = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ \mathsf{let}\ h = \mathsf{new}\ t\ \mathsf{in}\ (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} \end{array}\right\}_{\top}$$

Bind on $(\mathsf{let}\ h = []\ \mathsf{in}\ (push_1, pop_1))[\mathsf{new}\ \mathsf{inj}_1\ ()]$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S \mathsf{let}\ t = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ \mathsf{let}\ h = \mathsf{new}\ t\ \mathsf{in}\ (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} \end{array}\right\}_{\top}$$

Open R

// Let $K_{1S} \triangleq \mathsf{let}\ t = []\ \mathsf{in}\ \mathsf{let}\ h = \mathsf{new}\ t\ \mathsf{in}\ (push_2, pop_2)$

// Let $K_{2S} \triangleq \mathsf{let}\ h = \mathsf{new}\ t\ \mathsf{in}\ (push_2, pop_2)$

$$\left\{\begin{array}{l} j \overset{\zeta}{\Rightarrow}_S K_1[\mathsf{new}\ \mathsf{inj}_1\ ()] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}(h_0, e_0, \zeta) \end{array}\right\}_{\top \setminus R}$$

Frame

$$\{\textsc{Heap}\}_{\top \setminus R}$$

$$\mathsf{new}\ \mathsf{inj}_1\ ()$$

// Follows from Lemma .0.78

$$\{h_I.\ \textsc{Heap} * h_I \mapsto_I \mathsf{inj}_1\ ()\}_{\top \setminus R}$$

$$\left\{\begin{array}{l} h_I.\ j \overset{\zeta}{\Rightarrow}_S K_1[\mathsf{new}\ \mathsf{inj}_1\ ()][\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * h_I \mapsto_I \mathsf{inj}_1\ () \end{array}\right\}_{\top \setminus R}$$

// Follows from Lemma .0.79

$$\left\{\begin{array}{l} h_I.\ \exists l_S.\ j \overset{\zeta}{\Rightarrow}_S K_{1S}[l_S] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * h_I \mapsto_I \mathsf{inj}_1\ () * \\ l_S \mapsto^{\zeta}_S \mathsf{inj}_1\ () \end{array}\right\}_{\top \setminus R}$$

$$\left\{\begin{array}{l} h_I.\ \exists l_S.\ j \overset{\zeta}{\Rightarrow}_S K_{2S}[\mathsf{new}\ l_S] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * h_I \mapsto_I \mathsf{inj}_1\ () * \\ l_S \mapsto^{\zeta}_S \mathsf{inj}_1\ () \end{array}\right\}_{\top \setminus R}$$

// Follows from Lemma .0.79

$$\left\{\begin{array}{l} h_I.\ \exists h_S, l_S.\ j \overset{\zeta}{\Rightarrow}_S K_{2S}[h_S] * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * h_I \mapsto_I \mathsf{inj}_1\ () * \\ l_S \mapsto^{\zeta}_S \mathsf{inj}_1\ () * h_S \mapsto^{\zeta}_S l_S \end{array}\right\}_{\top \setminus R}$$

$$\left\{\begin{array}{l} h_I.\ \exists h_S, l_S.\ j \overset{\zeta}{\Rightarrow}_S (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * h_I \mapsto_I \mathsf{inj}_1\ () * l_S \mapsto^{\zeta}_S \mathsf{inj}_1\ () * h_S \mapsto^{\zeta}_S l_S \end{array}\right\}_{\top}$$

// Extending reg: Lemma .0.80

$$\left\{\begin{array}{l} h_I.\ \exists h_S, l_S.\ j \overset{\zeta}{\Rightarrow}_S (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * h_I \overset{1}{\hookrightarrow}_{I,r} \mathsf{inj}_1\ () * l_S \overset{1}{\hookrightarrow}_{S,r} \mathsf{inj}_1\ () * h_S \overset{1}{\hookrightarrow}_{S,r} l_S \end{array}\right\}_{\top}$$

// Fold into $\textsc{StackInv}((h_I, h_S), r, V[\![\tau]\!]^M)$ for the empty list by Lemma .0.112

$$\left\{\begin{array}{l} h_I.\ \exists h_S, l_S.\ j \overset{\zeta}{\Rightarrow}_S (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}((h_I, h_S), r, V[\![\tau]\!]^M) \end{array}\right\}_{\top}$$

$$\forall h_I.\ \left\{\begin{array}{l} \exists h_S, l_S.\ j \overset{\zeta}{\Rightarrow}_S (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Al}]^{\pi}_{r} * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}((h_I, h_S), r, V[\![\tau]\!]^M) \end{array}\right\}_{\top}$$

$(push_1, pop_1)$

$$\left\{\begin{array}{l} v_I^1.\ v_I^1 = (push_1, pop_1) * \exists v_S^1.\ j \overset{\zeta}{\Rightarrow}_S v_S^1 * v_S^1 = (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ P_{reg}(\{\rho\}, g, \{al_\rho\}, M, \zeta) * \textsc{StackInv}((h_I, h_S), r, V[\![\tau]\!]^M) \end{array}\right\}_{\top}$$

$$\left\{\begin{array}{l} v_I^1.\ \exists v_S^1.\ j \overset{\zeta}{\Rightarrow}_S v_S^1 * v_S^1 = (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{al_\rho\}, M, \zeta) * \\ \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * V[\![\tau \to^{\rho|\cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1}]\!]^M(push_1, push_2) * \\ V[\![\mathbf{1} \to^{\rho|\cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau]\!]^M(pop_1, pop_2) \end{array}\right\}_{\top}$$

$$\left\{\begin{array}{l} v_I^1.\ \exists v_S^1.\ j \overset{\zeta}{\Rightarrow}_S v_S^1 * v_S^1 = (push_2, pop_2) * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{al_\rho\}, M, \zeta) * \\ \textsc{StackInv}(h, r, V[\![\tau]\!]^M) * \\ V[\![\tau \to^{\rho|\cdot}_{wr_\rho, rd_\rho, al_\rho} \mathbf{1} \times \mathbf{1} \to^{\rho|\cdot}_{wr_\rho, rd_\rho} \mathbf{1} + \tau]\!]^M((push_1, pop_1), (push_2, pop_2)) \end{array}\right\}_{\top}$$

□

Example: Private Stacks

Consider the following two stack-modules:

$Stack_1$ has a single reference to a pure functional list, where the plain assignments updates the entire list on push and pop.

$$\begin{aligned} create_1() &= \mathsf{let}\ h = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ (push_1, pop_1) \\ push_1(n) &= \mathsf{let}\ v = !h\ \mathsf{in}\ h := \mathsf{inj}_2\ (n, v) \\ pop_1() &= \mathsf{let}\ v = !h\ \mathsf{in} \\ &\qquad \mathsf{case}(v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\ &\qquad\quad \mathsf{inj}_2\ (n, v') \Rightarrow h := v'; \mathsf{inj}_2\ n) \end{aligned}$$

$Stack_2$ has a single reference to a pure functional list, where the **cas** operation is used to update the entire list on push and pop.

$$\begin{aligned} create_2() &= \mathsf{let}\ h = \mathsf{new}\ \mathsf{inj}_1\ ()\ \mathsf{in}\ (push_2, pop_2) \\ push_2(n) &= \mathsf{let}\ v = !h\ \mathsf{in} \\ &\qquad \mathsf{let}\ v' = \mathsf{inj}_2\ (n, v)\ \mathsf{in}\ \mathsf{if}\ \mathsf{CAS}(h, v, v')\ \mathsf{then}\ ()\ \mathsf{else}\ push_2(n) \\ pop_2() &= \mathsf{let}\ v = !h\ \mathsf{in} \\ &\qquad \mathsf{case}(v, \mathsf{inj}_1\ () \Rightarrow \mathsf{inj}_1\ (), \\ &\qquad\quad \mathsf{inj}_2\ (n, v') \Rightarrow \mathsf{if}\ \mathsf{CAS}(h, v, v')\ \mathsf{then}\ \mathsf{inj}_2\ n\ \mathsf{else}\ pop_2()) \end{aligned}$$

This example shows, that if we know the module is private to us, we can directly update the value without the need for doing compare-and-swap.

We choose the following relation to show equality:

$$\begin{aligned}\textsc{StackRel}(h, r, \phi) &\triangleq ([\textsc{Wr}]^1_r \vee (\exists l, v_I, v_S.\ h_I \overset{1}{\hookrightarrow}_{I,r} v_I * h_S \overset{1}{\hookrightarrow}_{S,r} v_S * \\ &\qquad vals(l, (v_I, v_S), \phi)))\\ \textsc{StackInv}(h) &\triangleq \exists \iota.\ \boxed{\textsc{StackRel}(h, V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)}\end{aligned}$$

where

$$\begin{aligned}vals(\mathsf{nil}, v, \phi) &\triangleq v_I = \mathbf{inj}_1\ () \wedge v_S = \mathbf{inj}_1\ ()\\ vals(x :: xs, v, \phi) &\triangleq \exists v'_I, v'_S.\ v_I = \mathbf{inj}_2\ (x_I, v'_I) \wedge v_S = \mathbf{inj}_2\ (x_S, v'_S) \wedge \phi(x) \wedge \\ &\qquad vals(xs, (v'_I, v'_S), \phi)\end{aligned}$$

We only show the refinement proof of push, the proof of pop is straightforward.

Lemma .0.116. *$Stack_1$-push refines $Stack_2$-push*

$$\begin{aligned}&\forall \rho, M, h, n, m.\ V[\![\tau]\!]^M(n, m)\\ \Rightarrow\quad &\textsc{StackInv}(h, M(\rho)) \vdash E^{\rho;\cdot}_{wr_\rho, rd_\rho; M}(V[\![\mathbf{1}]\!]^M)(push_1(n), push_2(m))\end{aligned}$$

Proof. We define the following short-hands:

$$\begin{aligned}e_{1I} &\triangleq \mathbf{let}\ v = !h\ \mathbf{in}\ h := \mathbf{inj}_2\ (n, v)\\ e_{1S} &\triangleq \mathbf{let}\ v = !h_I\ \mathbf{in}\ \mathbf{let}\ v' = \mathbf{inj}_2\ (n, v)\ \mathbf{in}\ \mathbf{if}\ \mathsf{CAS}(h, v, v')\ \mathbf{then}\ ()\ \mathbf{else}\ push_2(n)\\ K_{1I} &\triangleq \mathbf{let}\ v = []\ \mathbf{in}\ h := \mathbf{inj}_2\ (n, v)\\ K_{1S} &\triangleq \mathbf{let}\ v = []\ \mathbf{in}\ \mathbf{let}\ v' = \mathbf{inj}_2\ (n, v)\ \mathbf{in}\ \mathbf{if}\ \mathsf{CAS}(h, v, v')\ \mathbf{then}\ ()\ \mathbf{else}\ push_2(n)\\ K_{2S} &\triangleq \mathbf{let}\ v' = []\ \mathbf{in}\ \mathbf{if}\ \mathsf{CAS}(h, v^1_I, v')\ \mathbf{then}\ ()\ \mathbf{else}\ push_1(n)\\ K_{3S} &\triangleq \mathbf{if}\ []\ \mathbf{then}\ ()\ \mathbf{else}\ push_2(n)\end{aligned}$$

Context: $g, j, K, \pi', \zeta, M, h, n$

Context: $\boxed{\textsc{Heap}}^{\textsc{Hp}}, \boxed{\textsc{Spec}(h_0, e_0, \zeta)}^{\textsc{Sp}(\zeta)}, \textsc{StackInv}(h, M(\rho)), V[\![\tau]\!]^M(n, n)$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) \right\}$

▷ $push(n)$

$\left\{ v^1_I.\ \exists v^1_S.\ j \overset{\zeta}{\Rightarrow}_S v^1_S * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) * V[\![\mathbf{1}]\!]^M(v^1_I, v^1_S) \right\}_{\top}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * P_{reg}(\{\rho\}, g, \{wr_\rho, rd_\rho\}, M, \zeta) \right\}_{\top}$

// Let $\pi = g(\rho)$, $r = M(\rho)$ and $R = \{\textsc{Hp}, \textsc{Sp}(\zeta), \textsc{Rg}(r)\}$

$\left\{ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} \right\}_{\top}$

Bind on $K_{11}[!h_I]$

// Unfolding $\textsc{StackInv}(h, r)$

$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \\ \exists \iota.\ \boxed{\textsc{StackRel}(h, r, V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)} \end{array} \right\}_{\top}$

Open $R, \textsc{Si}(\iota)$

$\left\{ \begin{array}{l} j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \triangleright \textsc{Reg}(r) * \\ [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \triangleright \textsc{Heap} * \triangleright \textsc{Spec}(h_0, e_0, \zeta) * \\ \triangleright \textsc{StackRel}(h, r, V[\![\tau]\!]^M) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$!h_I$

// Follows from Lemma .0.63

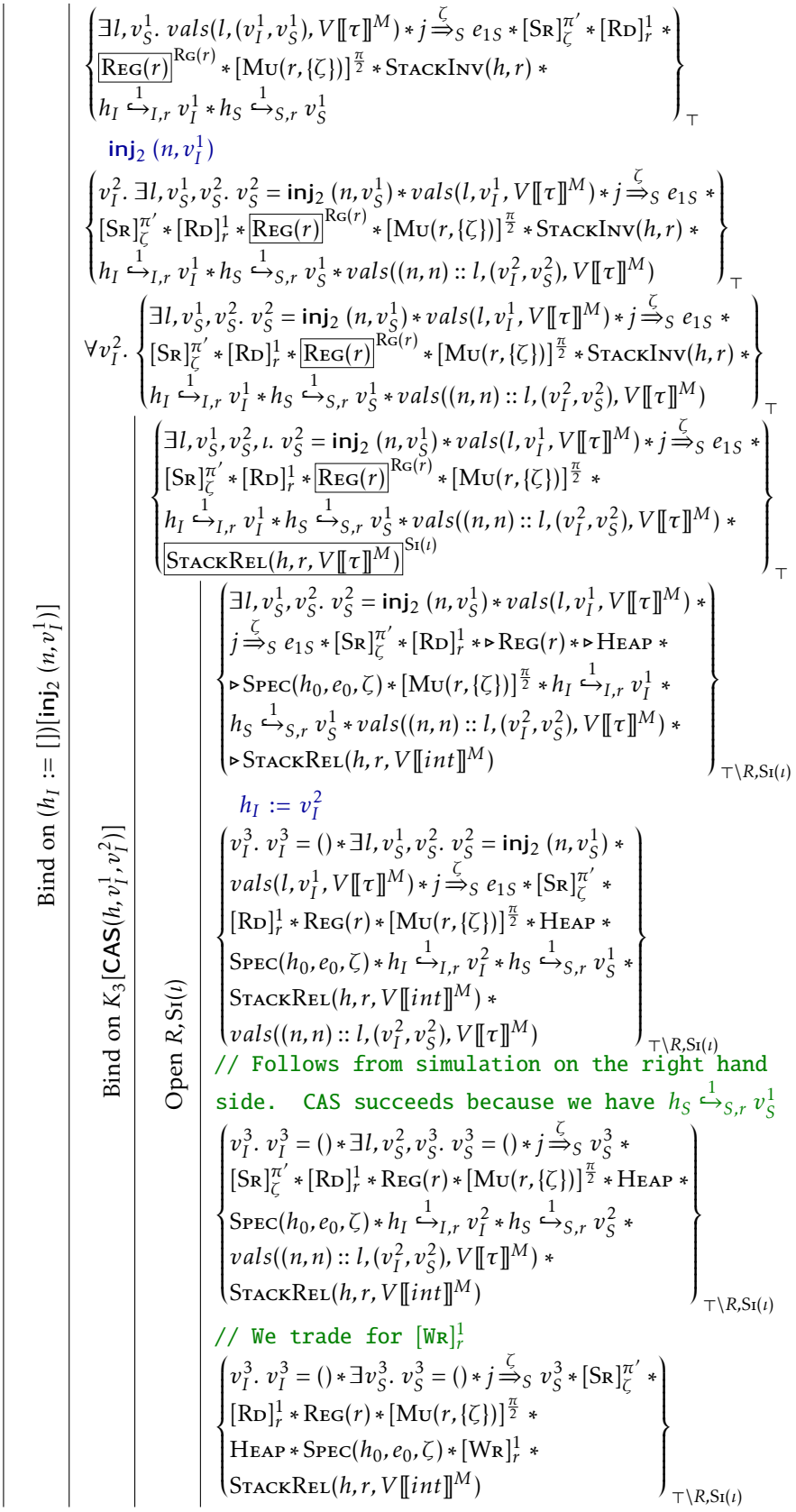
$\left\{ \begin{array}{l} v^1_I.\ \exists l, v^1_S.\ vals(l, (v^1_I, v^1_S), V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ [\textsc{Rd}]^1_r * [\textsc{Wr}]^1_r * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \\ \textsc{Spec}(h_0, e_0, \zeta) * \textsc{StackRel}(h, r, V[\![\tau]\!]^M) \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

// Trade $[\textsc{Wr}]^1_r$ in $\textsc{StackRel}(h, r, V[\![\tau]\!]^M)$

$\left\{ \begin{array}{l} v^1_I.\ \exists l, v^1_S.\ vals(l, (v^1_I, v^1_S), V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * \\ [\textsc{Rd}]^1_r * \textsc{Reg}(r) * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * \\ \textsc{StackRel}(h, r, V[\![\tau]\!]^M) * h_I \overset{1}{\hookrightarrow}_{I,r} v^1_I * h_S \overset{1}{\hookrightarrow}_{S,r} v^1_S \end{array} \right\}_{\top \setminus R, \textsc{Si}(\iota)}$

$\left\{ \begin{array}{l} v^1_I.\ \exists l, v^1_S.\ vals(l, (v^1_I, v^1_S), V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^1_r * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r) * \\ h_I \overset{1}{\hookrightarrow}_{I,r} v^1_I * h_S \overset{1}{\hookrightarrow}_{S,r} v^1_S \end{array} \right\}_{\top}$

$\forall v^1_I.$ $\left\{ \begin{array}{l} \exists l, v^1_S.\ vals(l, (v^1_I, v^1_S), V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]^{\pi'}_{\zeta} * [\textsc{Rd}]^1_r * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r, \{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h, r) * \\ h_I \overset{1}{\hookrightarrow}_{I,r} v^1_I * h_S \overset{1}{\hookrightarrow}_{S,r} v^1_S \end{array} \right\}_{\top}$


$\left\{\begin{array}{l}\exists l, v_S^1.\ vals(l,(v_I^1,v_S^1),V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * \\ \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h,r) * \\ h_I \overset{1}{\hookrightarrow}_{I,r} v_I^1 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1\end{array}\right\}_\top$
$\mathsf{inj}_2\ (n, v_I^1)$
$\left\{\begin{array}{l}v_I^2.\ \exists l, v_S^1, v_S^2.\ v_S^2 = \mathsf{inj}_2\ (n, v_S^1) * vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * \\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h,r) * \\ h_I \overset{1}{\hookrightarrow}_{I,r} v_I^1 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1 * vals((n,n)::l,(v_I^2,v_S^2),V[\![\tau]\!]^M)\end{array}\right\}_\top$
$\forall v_I^2.$ $\left\{\begin{array}{l}\exists l, v_S^1, v_S^2.\ v_S^2 = \mathsf{inj}_2\ (n, v_S^1) * vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * \\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{StackInv}(h,r) * \\ h_I \overset{1}{\hookrightarrow}_{I,r} v_I^1 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1 * vals((n,n)::l,(v_I^2,v_S^2),V[\![\tau]\!]^M)\end{array}\right\}_\top$
$\left\{\begin{array}{l}\exists l, v_S^1, v_S^2, \iota.\ v_S^2 = \mathsf{inj}_2\ (n, v_S^1) * vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * \\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \\ h_I \overset{1}{\hookrightarrow}_{I,r} v_I^1 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1 * vals((n,n)::l,(v_I^2,v_S^2),V[\![\tau]\!]^M) * \\ \boxed{\textsc{StackRel}(h,r,V[\![\tau]\!]^M)}^{\textsc{Si}(\iota)}\end{array}\right\}_\top$
$\left\{\begin{array}{l}\exists l, v_S^1, v_S^2.\ v_S^2 = \mathsf{inj}_2\ (n, v_S^1) * vals(l, v_I^1, V[\![\tau]\!]^M) * \\ j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * \triangleright \textsc{Reg}(r) * \triangleright \textsc{Heap} * \\ \triangleright \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^1 * \\ h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1 * vals((n,n)::l,(v_I^2,v_S^2),V[\![\tau]\!]^M) * \\ \triangleright \textsc{StackRel}(h,r,V[\![int]\!]^M)\end{array}\right\}_{\top\setminus R,\textsc{Si}(\iota)}$
$h_I := v_I^2$
$\left\{\begin{array}{l}v_I^3.\ v_I^3 = () * \exists l, v_S^1, v_S^2.\ v_S^2 = \mathsf{inj}_2\ (n, v_S^1) * \\ vals(l, v_I^1, V[\![\tau]\!]^M) * j \overset{\zeta}{\Rightarrow}_S e_{1S} * [\textsc{Sr}]_\zeta^{\pi'} * \\ [\textsc{Rd}]_r^1 * \textsc{Reg}(r) * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \\ \textsc{Spec}(h_0, e_0, \zeta) * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1 * \\ \textsc{StackRel}(h,r,V[\![int]\!]^M) * \\ vals((n,n)::l,(v_I^2,v_S^2),V[\![\tau]\!]^M)\end{array}\right\}_{\top\setminus R,\textsc{Si}(\iota)}$
// Follows from simulation on the right hand side. CAS succeeds because we have $h_S \overset{1}{\hookrightarrow}_{S,r} v_S^1$
$\left\{\begin{array}{l}v_I^3.\ v_I^3 = () * \exists l, v_S^2, v_S^3.\ v_S^3 = () * j \overset{\zeta}{\Rightarrow}_S v_S^3 * \\ [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * \textsc{Reg}(r) * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \textsc{Heap} * \\ \textsc{Spec}(h_0, e_0, \zeta) * h_I \overset{1}{\hookrightarrow}_{I,r} v_I^2 * h_S \overset{1}{\hookrightarrow}_{S,r} v_S^2 * \\ vals((n,n)::l,(v_I^2,v_S^2),V[\![\tau]\!]^M) * \\ \textsc{StackRel}(h,r,V[\![int]\!]^M)\end{array}\right\}_{\top\setminus R,\textsc{Si}(\iota)}$
// We trade for $[\textsc{Wr}]_r^1$
$\left\{\begin{array}{l}v_I^3.\ v_I^3 = () * \exists v_S^3.\ v_S^3 = () * j \overset{\zeta}{\Rightarrow}_S v_S^3 * [\textsc{Sr}]_\zeta^{\pi'} * \\ [\textsc{Rd}]_r^1 * \textsc{Reg}(r) * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \\ \textsc{Heap} * \textsc{Spec}(h_0, e_0, \zeta) * [\textsc{Wr}]_r^1 * \\ \textsc{StackRel}(h,r,V[\![int]\!]^M)\end{array}\right\}_{\top\setminus R,\textsc{Si}(\iota)}$
Open $R, \textsc{Si}(\iota)$
Bind on $K_3[\mathsf{CAS}(h, v_I^1, v_I^2)]$
Bind on $(h_I := [])[\mathsf{inj}_2\ (n, v_I^1)]$

$$\left\{\begin{array}{l} v_I^3.\ \exists v_S^3.\ j \overset{\zeta}{\Rightarrow}_S v_S^3 * [\textsc{Sr}]_\zeta^{\pi'} * [\textsc{Rd}]_r^1 * [\textsc{Wr}]_r^1 * \boxed{\textsc{Reg}(r)}^{\textsc{Rg}(r)} * [\textsc{Mu}(r,\{\zeta\})]^{\frac{\pi}{2}} * \\ \textsc{StackInv}(h) * V[\![\mathbf{1}]\!]^M(v_I^3, v_S^3) \end{array}\right\}_\top$$
$$\left\{v_I^3.\ \exists v_S^3.\ j \overset{\zeta}{\Rightarrow}_S v_S^3 * [\textsc{Sr}]_\zeta^{\pi'} * P_{reg}(\{\rho\}, g, \left\{wr_\rho, rd_\rho\right\}, M, \zeta) * V[\![\mathbf{1}]\!]^M(v_I^3, v_S^3)\right\}_\top$$

□

Bibliography

[1] Amal Ahmed.
Semantics of Types for Mutable State.
PhD thesis, Princeton University, 2004.
Cited on page 77.

[2] Amal Ahmed.
Step-indexed syntactic logical relations for recursive and quantified types.
In *European Symposium on Programming (ESOP)*, 2006.
Cited on page 72.

[3] Amal Ahmed, Derek Dreyer, and Andreas Rossberg.
State-dependent representation independence.
In *Principles of Programming Languages (POPL)*, 2009.
Cited on pages 33 and 106.

[4] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga.
A stratified semantics of general references embeddable in higher-order logic.
In *Logic in Computer Science (LICS)*, pages 75–86. IEEE Computer Society Press, 2002.
Cited on pages 30, 43, and 77.

[5] Andrew Appel and David McAllester.
An indexed model of recursive types for foundational proof-carrying code.
Programming Languages and Systems (TOPLAS), 23(5):657–683, 2001.
Cited on pages 78 and 118.

[6] Andrew Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon.
A very modal model of a modern, major, general type system.
In *Principles of Programming Languages (POPL)*, 2007.
Cited on pages 78 and 118.

[7] A.W. Appel, P-A. Melliès, C.D. Richards, and J. Vouillon.

A very modal model of a modern, major, general type system.
In *Principles of Programming Languages (POPL)*, 2007.
Cited on page 64.

[8] N. Benton, M. Hofmann, and V. Nigam.
Abstract effects and proof-relevant logical relations.
In *Principles of Programming Languages (POPL)*, 2014.
Cited on pages 11 and 63.

[9] N. Benton, M. Hofmann, and V. Nigam.
Effect-dependent transformations for concurrent programs.
In *Principles and Practice of Declarative Programming (PPDP)*, 2016.
Cited on pages 11, 12, 32, and 63.

[10] Nick Benton and Peter Buchlovsky.
Semantics of an effect analysis for exceptions.
In *International Workshop on Types in Language Design and Implementation (TLDI)*, 2007.
Cited on pages 9, 11, 37, 63, and 99.

[11] Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer.
Reading, writing and relations.
In *PLAS*. Springer, 2006.
Cited on pages 7 and 30.

[12] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann.
Relational semantics for effect-based program transformations with dynamic allocation.
In *Principles and Practice of Declarative Programming (PPDP)*, 2007.
Cited on pages 7 and 30.

[13] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann.
Relational semantics for effect-based program transformations: higher-order store.
In *Principles and Practice of Declarative Programming (PPDP)*, 2009.
Cited on pages 63 and 99.

[14] L. Birkedal, F. Sieczkowski, and J. Thamsborg.
A concurrent logical relation.
In *Computer Science Logic (CSL)*, 2012.
Cited on pages 8, 11, 12, 30, 32, 34, 35, and 63.

[15] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang.
Step-indexed Kripke models over recursive worlds.
In *Principles of Programming Languages (POPL)*, 2011.
Cited on pages 30, 43, and 77.

[16] Matko Botincan, Mike Dodds, and Suresh Jagannathan.
Proof-Directed Parallelization Synthesis by Separation Logic.
Programming Languages and Systems (TOPLAS), 35(2), 2013.
Cited on page 64.

[17] The Coq Development Team.
The Coq Proof Assistant Reference Manual, 2016.
URL http://coq.inria.fr.
Version 8.6.
Cited on page 140.

[18] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner.
TaDA: A logic for time and data abstraction.
In *European Conference on Object-Oriented Programming (ECOOP)*, pages 207–231, 2014.
Cited on pages 26, 140, 160, and 161.

[19] Pierre-Malo Deniélou and Nobuko Yoshida.
Dynamic multirole session types.
In *Principles of Programming Languages (POPL)*, volume 46, pages 435–446. ACM, 2011.
Cited on page 136.

[20] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang.
Views: compositional reasoning for concurrent programs.
In *Principles of Programming Languages (POPL)*, volume 48, pages 287–300. ACM, 2013.
Cited on page 137.

[21] D. Dreyer, A. Ahmed, and L. Birkedal.
Logical step-indexed logical relations.
Logical Methods in Computer Science (LMCS), 7(2:16), 2011.
Cited on pages 64, 78, and 118.

[22] Derek Dreyer, Georg Neis, and Lars Birkedal.
The impact of higher-order state and control effects on local relational reasoning.
In *International Conference on Functional Programming (ICFP)*, 2010.
Cited on page 106.

[23] M. Fähndrich and R. DeLine.
Adoption and focus: practical linear types for imperative programming.
In *Programming Language Design and Implementation (PLDI)*, 2002.
Cited on pages 7 and 30.

[24] Matthias Felleisen and Robert Hieb.
The revised report on the syntactic theories of sequential control and state.
Theoretical Computer Science (TCS), 103(2):235–271, 1992.
Cited on page 73.

[25] Robert W Floyd.
Assigning meanings to programs.
Mathematical aspects of computer science, 19(19-32):1, 1967.
Cited on page 106.

[26] D. K. Gifford and J. M. Lucassen.
Integrating functional and imperative programming.
In *LISP and functional programming (LISP)*, 1986.
Cited on pages 7, 8, 13, 30, and 99.

[27] James N Gray.
Notes on data base operating systems.
In *Operating Systems*, pages 393–481. Springer, 1978.
Cited on page 128.

[28] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill.
Ironfleet: proving practical distributed systems correct.
In *Symposium on Operating Systems Principles (SOSP)*, pages 1–17. ACM, 2015.
Cited on pages 17, 23, 104, 134, and 136.

[29] Maurice P. Herlihy and Jeannette M. Wing.
Linearizability: a correctness condition for concurrent objects.
Programming Languages and Systems (TOPLAS), 12(3):463–492, July 1990.
Cited on page 140.

[30] Gerard J. Holzmann.
The model checker spin.
IEEE Transactions on software engineering, 23(5):279–295, 1997.
Cited on pages 17, 104, and 136.

[31] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo.
Language primitives and type discipline for structured communication-based programming.
In *European Symposium on Programming (ESOP)*, pages 122–138. Springer, 1998.
Cited on page 136.

[32] Kohei Honda, Nobuko Yoshida, and Marco Carbone.
Multiparty asynchronous session types.

Principles of Programming Languages (POPL), 43(1):273–284, 2008.
Cited on page 136.

[33] 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 *Principles of Programming Languages (POPL)*, pages 637–650, 2015.
Cited on pages 30, 44, 64, 66, 77, 106, 117, 140, and 160.

[34] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer.
Higher-order ghost state.
In *International Conference on Functional Programming (ICFP)*, pages 256–269, 2016.
Cited on pages 66, 77, and 106.

[35] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer.
Rustbelt: Securing the foundations of the rust programming language.
Principles of Programming Languages (POPL), 2017.
Cited on page 108.

[36] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis.
Strong logic for weak memory: Reasoning about release-acquire consistency in iris.
In *European Conference on Object-Oriented Programming (ECOOP)*, volume 74, 2017.
Cited on page 108.

[37] Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat.
Mace: language support for building distributed systems.
In *Programming Language Design and Implementation (PLDI)*, volume 42, pages 179–188. ACM, 2007.
Cited on pages 17, 104, and 136.

[38] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal.
The essence of higher-order concurrent separation logic.
In *European Symposium on Programming (ESOP)*, April 2017.
Cited on pages 17, 66, 78, 79, 93, 97, 104, 106, 117, 118, 120, 137, and 140.

[39] Robbert Krebbers, Amin Timany, and Lars Birkedal.
Interactive Proofs in Higher-Order Concurrent Separation Logic.
In *Principles of Programming Languages (POPL)*, 2017.
Cited on pages 66, 77, 97, 98, 100, 106, and 140.

[40] N. Krishnaswami, P. Pradic, and N. Benton.
Integrating linear and dependent types.
In *Principles of Programming Languages (POPL)*, 2015.
Cited on pages 7 and 30.

[41] Morten Krogh-Jespersen, Thomas Dinsdale-Young, and Lars Birkedal.
Verifying a Concurrent Data-Structure from the Dartino Framework, 2017.
Cited on pages 24, 25, and 26.

[42] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal.
A relational model of types-and-effects in higher-order concurrent separation logic.
In *Principles of Programming Languages (POPL)*, pages 218–231, 2017.
Cited on pages 3, 7, 8, 9, 11, 16, 66, 77, 100, and 101.

[43] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, and Lars Birkedal.
Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems, 2018.
Cited on pages 17, 22, and 23.

[44] Leslie Lamport.
Proving the correctness of multiprocess programs.
IEEE transactions on software engineering, (2):125–143, 1977.
Cited on page 106.

[45] Leslie Lamport.
The implementation of reliable distributed multiprocess systems.
Computer networks, 2(2):95–114, 1978.
Cited on page 106.

[46] Leslie Lamport.
Hybrid systems in tla+.
In *Hybrid Systems*, pages 77–102. Springer, 1993.
Cited on pages 17, 104, and 136.

[47] John Launchbury and Simon Peyton Jones.
Lazy functional state threads.
In *Programming Language Design and Implementation (PLDI)*, 1994.
Cited on pages 12, 13, 16, 66, 67, 68, 99, and 101.

[48] John Launchbury and Simon L. Peyton Jones.
State in haskell.
Lisp and symbolic computation, 8(4):293–341, 1995.
Cited on pages 13, 16, 66, and 69.

[49] K Rustan M Leino.
Dafny: An automatic program verifier for functional correctness.
In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, pages 348–370. Springer, 2010.
Cited on pages 23 and 136.

[50] Mohsen Lesani, Christian J Bell, and Adam Chlipala.
Chapar: certified causally consistent distributed key-value stores.
In *Principles of Programming Languages (POPL)*, volume 51, pages 357–370. ACM, 2016.
Cited on pages 17, 104, 134, and 136.

[51] John M Lucassen and David K Gifford.
Polymorphic effect systems.
In *Principles of Programming Languages (POPL)*, 1988.
Cited on pages 7, 8, and 30.

[52] E. Moggi and Amr Sabry.
Monadic encapsulation of effects: A revised approach (extended version).
Journal of Functional Programming, 11(6):591–627, November 2001.
ISSN 0956-7968.
Cited on pages 13, 16, 17, 66, and 99.

[53] G. Morrisett, A. Ahmed, and M. Fluet.
L^3: A linear language with locations.
In *Typed Lambda Calculi and Applications (TLCA)*, 2005.
Cited on pages 7 and 30.

[54] A. Nanevski, G. Morrisett, and L. Birkedal.
Polymorphism and separation in hoare type theory.
In *International Conference on Functional Programming (ICFP)*, 2006.
Cited on pages 7 and 30.

[55] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco.
Communicating state transition systems for fine-grained concurrent resources.
In *European Symposium on Programming (ESOP)*, pages 290–310, 2014.
Cited on page 137.

[56] Peter W O'hearn.
Resources, concurrency, and local reasoning.
Theoretical Computer Science (TCS), 375(1-3):271–307, 2007.
Cited on pages 20, 108, and 137.

[57] Peter O'Hearn, John Reynolds, and Hongseok Yang.

Local reasoning about programs that alter data structures.
In *Computer Science Logic (CSL)*, pages 1–19. Springer, 2001.
Cited on page 140.

[58] G.D. Plotkin and M. Abadi.
A logic for parametric polymorphism.
In *Typed Lambda Calculi and Applications (TLCA)*, 1993.
Cited on page 64.

[59] Amir Pnueli.
The temporal logic of programs.
In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.
Cited on pages 104 and 136.

[60] F. Pottier.
Hiding local state in direct style: a higher-order anti-frame rule.
In *Logic in Computer Science (LICS)*, 2008.
Cited on pages 7 and 30.

[61] F. Pottier and J. Protzenko.
Programming with permissions in Mezzo.
In *International Conference on Functional Programming (ICFP)*, 2013.
Cited on pages 7, 30, and 31.

[62] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L Constable.
Formal specification, verification, and implementation of fault-tolerant systems using eventml.
Electronic Communications of the EASST, 72, 2015.
Cited on pages 17, 104, 134, and 137.

[63] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L Constable.
Eventml: Specification, verification, and implementation of crash-tolerant state machine replication systems.
Science of Computer Programming, 148:26–48, 2017.
Cited on page 137.

[64] Mohammad Raza, Cristiano Calcagno, and Philippa Gardner.
Automatic Parallelization with Separation Logic.
In *European Symposium on Programming (ESOP)*, 2009.
Cited on page 64.

[65] John C. Reynolds.
Types, abstraction, and parametric polymorphism.
Information Processing, 1983.
Cited on pages 15 and 80.

[66] John C Reynolds.
Separation logic: A logic for shared mutable data structures.
In *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
Cited on pages 107, 140, and 158.

[67] Rust Language.
https://doc.rust-lang.org, 2016.
Cited on pages 7, 30, and 31.

[68] Steven Schäfer, Tobias Tebbi, and Gert Smolka.
Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions.
In *Interactive Theorem Proving (ITP)*, volume 9236 of *LNCS*, pages 359–374, 2015.
Cited on page 98.

[69] Miley Semmelroth and Amr Sabry.
Monadic encapsulation in ml.
In *International Conference on Functional Programming (ICFP)*, pages 8–17, New York, NY, USA, 1999. ACM.
ISBN 1-58113-111-9.
Cited on pages 13 and 99.

[70] Ilya Sergey, James R Wilcox, and Zachary Tatlock.
Programming and proving with distributed protocols.
Principles of Programming Languages (POPL), 2:28, 2017.
Cited on pages 17, 22, 23, 104, 106, 134, 136, and 138.

[71] F. Smith, D. Walker, and G. Morrisett.
Alias types.
In *European Symposium on Programming (ESOP)*, 2000.
Cited on pages 7 and 30.

[72] Kasper Svendsen and Lars Birkedal.
Impredicative concurrent abstract predicates.
In *European Symposium on Programming Languages (ESOP)*, pages 149–168. Springer, 2014.
Cited on pages 106 and 137.

[73] Andrew S Tanenbaum and Maarten Van Steen.
Distributed systems: principles and paradigms.
Prentice-Hall, 2007.
Cited on page 129.

[74] Jacob Thamsborg and Lars Birkedal.
A kripke logical relation for effect-based program transformations.
In *International Conference on Functional Programming (ICFP)*, 2011.
Cited on pages 9, 11, 37, 63, 99, and 100.

[75] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal.
A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runst.
Principles of Programming Languages (POPL), page 64, 2017.
Cited on pages 3, 4, 13, 14, 15, 16, 17, and 108.

[76] Mads Tofte and Jean-Pierre Talpin.
Implementation of the typed call-by-value λ-calculus using a stack of regions.
In *Principles of Programming Languages (POPL)*, 1994.
Cited on pages 9 and 37.

[77] Bernardo Toninho, Luís Caires, and Frank Pfenning.
Dependent session types via intuitionistic linear type theory.
In *Principles and Practice of Declarative Programming (PPDP)*, pages 161–172. ACM, 2011.
Cited on page 136.

[78] Aaron Turon, Derek Dreyer, and Lars Birkedal.
Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency.
In *International Conference on Functional Programming (ICFP)*, pages 377–390, 2013.
Cited on pages 10, 50, and 64.

[79] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer.
Logical relations for fine-grained concurrency.
In *Principles of Programming Languages (POPL)*, 2013.
Cited on page 33.

[80] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer.
GPS: navigating weak memory with ghosts, protocols, and separation.
In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 691–707, 2014.
Cited on page 137.

[81] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson.
Verdi: a framework for implementing and formally verifying distributed systems.
In *Principles of Programming Languages (POPL)*, volume 50, pages 357–368. ACM, 2015.
Cited on pages 17, 23, 104, 134, and 137.