A Logical Relation for Monadic Encapsulation of State

Proving contextual equivalences in the presence of runST

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

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.

Additional Key Words and Phrases: ST Monad, Logical Relations, Functional Programming Languages, Theory of Programming Languages, Program Logics, Iris

ACM Reference format:
Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A Logical Relation for Monadic Encapsulation of State. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2017), 28 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 Peyton Jones [1994]. The ST monad comes equipped with a function runST : $(\forall \beta, \mathsf{ST}\ \beta\ \tau) \to \tau$ that allows a value to escape from the monad: runST runs a stateful computation of the monadic type $\mathsf{ST}\ \beta\ \tau$ and then returns the resulting value of type τ. In the original paper [Launchbury and Peyton Jones 1994], 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

[*] Most of this work was done while this author was at ENS Lyon

2017. 2475-1421/2017/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [Launchbury and Peyton Jones 1995; Moggi and Sabry 2001]. Note that, unlike Haskell (and [Moggi and Sabry 2001]) which is call-by-need, STLang is call-by-value. This choice simplifies one aspect the problem, because with a call-by-need semantics, one has to keep track of the dependencies between the thunks, and the evaluation of a term jumps around to evaluate its dependencies.

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 [Jung et al. 2016, 2015; Krebbers et al. 2017a]. Iris's base logic [Krebbers et al. 2017a] 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 [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017], 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 §3 and §5.

Another reason for using Iris is that the newly developed powerful proof mode for Iris [Krebbers et al. 2017b] 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 give a brief recap of the Haskell ST monad and why runST intuitively encapsulates state. Finally, we give an overview of the technical development and our new results.

1.1 A Recap of the Haskell ST Monad

The ST monad, as described in [Launchbury and Peyton Jones 1994] 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:

[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 }
```

Fig. 1. 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.

```
(==) :: 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.

Figure 1 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 βInteger)` and thus `fibST` indeed has return type **`Integer`**.

1.2 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 Peyton Jones [1994] 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 $\forall\beta$. ST β τ, 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:

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

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 [1995] is denotational and does not use a global mutable heap with in-place update, and they state [Launchbury and Peyton Jones 1995, 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 §7.

1.3 Overview of Results and the Technical Development

In §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 $\Xi \mid \Gamma \vDash e \preceq_{\mathrm{ctx}} e' : \tau$. As usual, e and e' are contextually equivalent, denoted $\Xi \mid \Gamma \vDash e \approx_{\mathrm{ctx}} e' : \tau$, 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 2. 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.

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

$$
\begin{array}{rcll}
e & \preceq_{\mathrm{ctx}} & () \ : \ \mathbf{1} & \text{(\textsc{Neutrality})}\\
\texttt{let}\, x = e_2 \,\texttt{in}\, (e_1, x) & \approx_{\mathrm{ctx}} & (e_1, e_2) \ : \ \tau_1 \times \tau_2 & \text{(\textsc{Commutativity})}\\
\texttt{let}\, x = e \,\texttt{in}\, (x, x) & \approx_{\mathrm{ctx}} & (e, e) \ : \ \tau \times \tau & \text{(\textsc{Idempotency})}\\
\texttt{let}\, y = e_1 \,\texttt{in}\, \texttt{rec}\, f(x) = e_2 & \preceq_{\mathrm{ctx}} & \texttt{rec}\, f(x) = \texttt{let}\, y = e_1 \,\texttt{in}\, e_2 \ : \ \tau_1 \to \tau_2 & \text{(\textsc{Rec hoisting})}\\
\texttt{let}\, y = e_1 \,\texttt{in}\, \Lambda\, e_2 & \preceq_{\mathrm{ctx}} & \Lambda\, (\texttt{let}\, y = e_1 \,\texttt{in}\, e_2) \ : \ \forall X.\, \tau & \text{(\textsc{Λ hoisting})}\\
e & \preceq_{\mathrm{ctx}} & \texttt{rec}\, f(x) = (e\ x) \ : \ \tau_1 \to \tau_2 & \text{(\textsc{η expansion for rec})}\\
e & \preceq_{\mathrm{ctx}} & \Lambda\, (e\ \_) \ : \ \forall X.\, \tau & \text{(\textsc{η expansion for Λ})}\\
(\texttt{rec}\, f(x) = e_1)\ e_2 & \preceq_{\mathrm{ctx}} & e_1[e_2, (\texttt{rec}\, f(x) = e_1)/x, f] \ : \ \tau & \text{(\textsc{β reduction for rec})}\\
(\Lambda\, e)\ \_ & \approx_{\mathrm{ctx}} & e : \tau[\tau'/X] & \text{(\textsc{β reduction for Λ})}
\end{array}
$$

Fig. 2. Contextual Refinements and Equivalences for Pure Computations

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 3.

$$
\begin{array}{rll}
\texttt{bind}\, e \,\texttt{in}\, (\lambda\, x.\, \texttt{return}\, x) \approx_{\mathrm{ctx}} & e : \mathsf{ST}\ \rho\ \tau & \text{(\textsc{Left Identity})}\\
e_2\ e_1 \preceq_{\mathrm{ctx}} & \texttt{bind}\, (\texttt{return}\, e_1) \,\texttt{in}\, e_2 : \mathsf{ST}\ \rho\ \tau & \text{(\textsc{Right Identity})}\\
\texttt{bind}\, (\texttt{bind}\, e_1 \,\texttt{in}\, e_2) \,\texttt{in}\, e_3 \preceq_{\mathrm{ctx}} & \texttt{bind}\, e_1 \,\texttt{in}\, (\lambda\, x.\, \texttt{bind}\, (e_2\ x) \,\texttt{in}\, e_3) : \mathsf{ST}\ \rho\ \tau' & \text{(\textsc{Associativity})}
\end{array}
$$

Fig. 3. Contextual Equivalences for Stateful Computations

The results in Figure 2 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 §3. We further use our logical relation to show the following State-Independence theorem:

Theorem 1.1 (State Independence).

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

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 §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 §2. There, we also formally state our definition of contextual refinement and contextual equivalence. In §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 to the precise statement and proof sketches of the refinements in Figures 2 and 3. In §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 §6. We discuss related work in §7 and conclude in §8.

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 4. Note that there are no types in the terms; following [Ahmed 2006] 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 $\mathtt{ref}(e)$ creates a new reference, !e reads from one and e ← 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.

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

Fig. 4. The syntax of STLang.

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

$$\textsc{TVar}\quad \Xi \mid \Gamma, x : \tau \vdash x : \tau \qquad \textsc{TRec}\quad \frac{\Xi \mid \Gamma, x : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2}{\Xi \mid \Gamma \vdash \mathtt{rec}\, f(x) = e : \tau_1 \to \tau_2} \qquad \textsc{TAbs}\quad \frac{\Xi, X \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall X.\, \tau}$$

$$\textsc{TFold}\quad \frac{\Xi \mid \Gamma \vdash e : \tau[\mu X.\, \tau / X]}{\Xi \mid \Gamma \vdash \mathtt{fold}\, e : \mu X.\, \tau} \qquad \textsc{TInst}\quad \frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]} \qquad \textsc{TNew}\quad \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \mathtt{ref}(e) : \mathsf{ST}\ \rho\ (\mathsf{STRef}\ \rho\ \tau)}$$

$$\textsc{TDeref}\quad \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash !\, e : \mathsf{ST}\ \rho\ \tau} \qquad \textsc{TGets}\quad \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : \mathsf{ST}\ \rho\ \mathbf{1}}$$

$$\textsc{TRefEq}\quad \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \mathsf{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash e == e' : \mathbb{B}} \qquad \textsc{TReturn}\quad \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \mathtt{return}\, e : \mathsf{ST}\ \rho\ \tau}$$

$$\textsc{TBind}\quad \frac{\Xi \mid \Gamma \vdash e : \mathsf{ST}\ \rho\ \tau \qquad \Xi \mid \Gamma \vdash e' : \tau \to (\mathsf{ST}\ \rho\ \tau')}{\Xi \mid \Gamma \vdash \mathtt{bind}\, e\, \mathtt{in}\, e' : \mathsf{ST}\ \rho\ \tau'} \qquad \textsc{TRunST}\quad \frac{\Xi, X \mid \Gamma \vdash e : \mathsf{ST}\ X\ \tau \qquad \Xi \vdash \tau}{\Xi \mid \Gamma \vdash \mathtt{runST}\ \{e\} : \tau}$$

Fig. 5. An excerpt of the typing rules for 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 5.

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 $h \in Loc \rightharpoonup^{\text{fin}} Val$ is a finite partial function that associates values to locations, which we suppose are positive integers ($Loc \triangleq \mathbb{Z}^+$)[2].

The semantics, shown in Figure 6, is presented in the Felleisen-Hieb style [Felleisen and Hieb 1992], using evaluation contexts K: 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

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

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

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

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:

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

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

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

$$\langle h, \mathtt{match}\, \mathtt{inj}_i\, v\, \mathtt{with}\, \mathtt{inj}_i\, x \Rightarrow e_i\ \mathtt{end}\rangle \to_h \langle h, e_i[v/x]\rangle \qquad \frac{\ell \neq \ell'}{\langle h, \ell == \ell'\rangle \to_h \langle h, \mathtt{false}\rangle}$$

$$\frac{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}{\langle h, \mathtt{runST}\ \{v\}\rangle \to_h \langle h', \mathtt{runST}\ \{e\}\rangle} \qquad \langle h, \mathtt{runST}\ \{\mathtt{return}\, v\}\rangle \to_h \langle h, v\rangle$$

Effectful reduction: $\boxed{\langle h, v\rangle \rightsquigarrow \langle h', e\rangle}$ and effectful head step: $\boxed{\langle h, v\rangle \rightsquigarrow_h \langle h', e\rangle}$
Effectful evaluation contexts: $\mathbb{K} ::= [] \mid \mathtt{bind}\, \mathbb{K}\, \mathtt{in}\, v$

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

Alloc

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

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

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

Fig. 6. An excerpt of the dynamics of STLang, a call-by-value, small-step operational semantics.

above can be a reducible expression, which is reduced by using $K = \mathtt{runST}\{[]\}$ as a context for the main reduction rule $\to$.

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.

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

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

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

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 $K = \texttt{runST}\ \{[]\}$ 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 2.1 (Contextual refinement and equivalence). We define contextual refinement $\preceq_{\text{ctx}}$ and contextual equivalence $\approx_{\text{ctx}}$ as follows.

$$\begin{aligned} \Xi \mid \Gamma \vDash e \preceq_{\text{ctx}} e' : \tau \ &\triangleq\ \Xi \mid \Gamma \vdash e : \tau \ \wedge\ \Xi \mid \Gamma \vdash e' : \tau\ \wedge \\ &\qquad \forall h, h', C.\ \ C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; \mathbf{1})\ \wedge\ (h, C[e])\!\downarrow\ \implies\ (h', C[e'])\!\downarrow \\ \Xi \mid \Gamma \vDash e \approx_{\text{ctx}} e' : \tau \ &\triangleq\ \Xi \mid \Gamma \vDash e \preceq_{\text{ctx}} e' : \tau\ \wedge\ \Xi \mid \Gamma \vDash e' \preceq_{\text{ctx}} e : \tau. \end{aligned}$$

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

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 [Ahmed 2004; Ahmed et al. 2002; Birkedal et al. 2011]. 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 [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017]. 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 [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017] 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 [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017], 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 §5.

3.1 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 [Jung et al. 2015]. Subsequently, Iris was extended with a notion of higher-order ghost state [Jung et al. 2016], 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 [Krebbers et al. 2017a].

In Iris one can quantify over the Iris types κ:

$$\kappa ::= \mathbf{1} \,|\, \kappa \times \kappa \,|\, \kappa \longrightarrow \kappa \,|\, \mathit{Expr} \,|\, \mathit{Val} \,|\, \mathbb{Z} \,|\, \mathbb{B} \,|\, \kappa \overset{\text{fin}}{\rightharpoonup} \kappa \,|\, \mathsf{finset}(\kappa) \,|\, \mathit{Monoid} \,|\, \mathit{Names} \,|\, \mathit{iProp} \,|\, \ldots$$

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, $\kappa \rightharpoonup^{\text{fin}} \kappa$ 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:

$$\begin{aligned} P ::= &\ \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \\ &\mid \triangleright P \mid \mu r.P \mid \ \square\, P \mid {\models\!\!\!\Rrightarrow} P \end{aligned}$$

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 $\kappa \longrightarrow \mathit{iProp}$ (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 [Appel and McAllester 2001; Appel et al. 2007; Dreyer et al. 2011]. In terms of step-indexing $\triangleright P$ holds if P holds a step later; hence the name. In Iris it can be used to define weakest preconditions and to guard impredicative invariants to avoid self-referential paradoxes [Krebbers et al. 2017a]. 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 [Dreyer et al. 2011]. 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 "always" modality ($\square$). 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 $P \vdash \square P$. Intuitively, $\square P$ holds if P holds without asserting any exclusive ownership. Hence $\square P$ is a duplicable assertion, i.e., we have $(\square P) * (\square P) \dashv\vdash \square P$, where $\dashv\vdash$ is the logical equivalence of formulas. Hence persistent propositions are therefore duplicable. The always modality is idempotent, $\square P \vdash \square \square P$, and also satisfies $\square P \vdash P$. The always modality (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] ($\mathrel{|\!\!\Rrightarrow}$). Intuitively, the proposition $\mathrel{|\!\!\Rrightarrow} P$ 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 $P \mathrel{\Rrightarrow\!\!*} Q$ as a shorthand for $P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow} Q$. The update modality is idempotent, $\mathrel{|\!\!\Rrightarrow}(\mathrel{|\!\!\Rrightarrow} P) \dashv\vdash \mathrel{|\!\!\Rrightarrow} P$.

3.2 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* $\mathrel{|\!\!\gg}\{\cdot\}\mathrel{\Rrightarrow\!\!\triangleright}$ as follows:

$$\mathrel{|\!\!\gg}\{n\}\mathrel{\Rrightarrow\!\!\triangleright} P \triangleq (\mathrel{|\!\!\Rrightarrow} \triangleright)^n \mathrel{|\!\!\Rrightarrow} P$$

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

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

$$\mathsf{IC}^{\gamma}\ e\ \{\!|v.\ Q|\!\} \triangleq \forall h_1, h_2, v, n.\ \langle h_1, e\rangle \to^n \langle h_2, v\rangle * heap_{\gamma}(h_1) \gg\{n\}\mathrel{\Rrightarrow\!\!*} heap_{\gamma}(h_2) * Q\ v$$

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 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 §5; here we just highlight the properties that these abstract predicates enjoy:

$$heap_{\gamma}(h) * \ell \mapsto_{\gamma} v \Rightarrow h(l) = v \tag{1}$$

$$heap_{\gamma}(h) \wedge l \notin \mathrm{dom}(h) \mathrel{\Rrightarrow\!\!*} heap_{\gamma}(h[l \mapsto v]) * \ell \mapsto_{\gamma} v \tag{2}$$

$$heap_{\gamma}(h) * \ell \mapsto_{\gamma} v \mathrel{\Rrightarrow\!\!*} heap_{\gamma}(h[l \mapsto v']) * \ell \mapsto_{\gamma} v' \tag{3}$$

$$\ell \mapsto_{\gamma} v * \ell \mapsto_{\gamma} v' \Rightarrow \bot \tag{4}$$

Property (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 (2) expresses that we can allocate a new location l in h, if l is not already in the domain of h. Finally, Property (3) says that we can update

[3] In [Krebbers et al. 2017a] 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 [Krebbers et al. 2017a].

the value at location l, if we have both $heap_\gamma(h)$ and $\ell \mapsto_\gamma v$. Property (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:

$$\{\!|P|\!\}\; e\; \{\!|v.\ Q|\!\}_\gamma \triangleq \square(P \twoheadrightarrow \mathsf{IC}^\gamma\; e\; \{\!|v.\ Q|\!\})$$

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.

3.3 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 7. 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 $(\mathit{Val} \times \mathit{Val}) \rightarrow \mathit{iProp}$ 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 [Reynolds 1983].

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:

$$\begin{aligned}[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v, v') \triangleq \exists w_1, w_2, w_1', w_2'.\ & v = (w_1, w_2) \wedge v' = (w_1', w_2') \wedge \\ & [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') \wedge [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')\end{aligned}$$

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 $\tau \rightarrow \tau'$ 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:

$$[\![\Xi \vdash \tau \rightarrow \tau']\!]_\Delta(v, v') \triangleq \square\left(\forall (w, w').\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow \mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(v\ w, v'\ w')\right).$$

The $\square$ modality is used to ensure that $[\![\Xi \vdash \tau \rightarrow \tau']\!]_\Delta(v, v')$ 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):

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

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

Value relations:

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

Expression relation:

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

Environment relation:

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

Logical relatedness:

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

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

Fig. 7. Binary logical relation

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).

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

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':

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

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:

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

The requirement that the reduction on the specification side is *deterministic* is used crucially in the proofs of the purity properties in §4. 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 ($\mathit{heap}_{\gamma'}(h_1')$), 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:

$$\mathsf{regions} \;\Rrightarrow\!\!\ast\; \exists r.\ \mathsf{region}(r, \gamma_h, \gamma_h') \tag{5}$$

Notice that (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 $\mathtt{STRef}\ \rho\ \tau$ 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 $[\![\Xi \vdash \mathtt{ST}\ \rho\ \tau]\!]_\Delta$ 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 $\langle h_1', \mathtt{runST}\ \{v'\}\rangle \to_d^* \langle h_2', w'\rangle$ 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 7.

3.4 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 3.1 (Properties of IC).

(1) $IC^\gamma\ e\ \{\!|v.\ Q|\!\} \ast (\forall w.\ (Q\ w) \mathrel{-\!\!\ast} IC^\gamma\ K[w]\ \{\!|v.\ Q'\ v|\!\}) \vdash IC^\gamma\ K[e]\ \{\!|v.\ Q'|\!\}$

(2) $\mid\!\!\Rrightarrow (Q\ w) \vdash IC^\gamma\ w\ \{\!|v.\ Q|\!\}$

(3) $(\forall v.\ (P\ v) \Rrightarrow\!\!\ast (Q\ v)) \ast IC^\gamma\ e\ \{\!|v.\ P|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$

(4) $\mid\!\!\Rrightarrow IC^\gamma\ e\ \{\!|v.\ Q|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$

(5) $IC^\gamma\ e\ \{\!|v.\ \mid\!\!\Rrightarrow Q|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$

(6) $(\forall h.\ \langle h, e\rangle \to \langle h, e'\rangle) \ast \triangleright IC^\gamma\ e'\ \{\!|v.\ Q|\!\} \vdash IC^\gamma\ e\ \{\!|v.\ Q|\!\}$

(7) $\triangleright(\forall \ell.\ \ell \mapsto_\gamma v \equiv\!\!* Q\ \ell) \vdash \mathit{IC}^\gamma\ \texttt{runST}\ \{\texttt{ref}(v)\}\ \{\!\!|w.\ Q|\!\!\}$

(8) $\triangleright \ell \mapsto_\gamma v * \triangleright(\ell \mapsto_\gamma v \equiv\!\!* Q\ v) \vdash \mathit{IC}^\gamma\ \texttt{runST}\ \{!\,\ell\}\ \{\!\!|w.\ Q|\!\!\}$

(9) $\triangleright \ell \mapsto_\gamma v' * \triangleright(\ell \mapsto_\gamma v \equiv\!\!* Q\ ()) \vdash \mathit{IC}^\gamma\ \texttt{runST}\ \{\ell \leftarrow v\}\ \{\!\!|w.\ Q|\!\!\}$

(10) $\mathit{IC}^\gamma\ \texttt{runST}\ \{e\}\ \{\!\!|v.\ Q|\!\!\} * \Big(\forall w.\ (Q\ w) -\!\!*$
$\mathit{IC}^\gamma\ \texttt{runST}\ \{\mathbb{K}[\texttt{return}\, w]\}\ \{\!\!|v.\ Q'\ w|\!\!\}\Big) \vdash \mathit{IC}^\gamma\ \texttt{runST}\ \{\mathbb{K}[e]\}\ \{\!\!|v.\ Q'|\!\!\}$

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 $\mid\!\Rrightarrow$ 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 3.2 (Compatibility for runST). *Suppose* $\Xi, X \mid \Gamma \vDash e \preceq_{\log} e' : \mathsf{ST}\ X\ \tau$ *and* $\Xi \vdash \tau$. *Then*

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

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 $\mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(\texttt{runST}\ \{v\}\,, \texttt{runST}\ \{v'\})$. 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 $\mathit{heap}_{\gamma'_h}(h'_1)$ for some h'_1. We are to show:

$$\mathsf{IC}^{\gamma_h}\ \texttt{runST}\ \{v\}\ \Big\{\!\!\Big|w.\ \exists h'_2, w'.\ \big\langle h'_1, \texttt{runST}\ \{v'\}\big\rangle \to_d^* \big\langle h'_2, w'\big\rangle * \mathit{heap}_{\gamma'_h}(h'_2) * [\![\Xi \vdash \tau]\!]_\Delta(w, w')\Big|\!\!\Big\}$$

Using (5) with regions we know there exists a fresh semantic region r and that the predicate $\mathrm{region}(r, \gamma_h, \gamma'_h)$ 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 $\mathit{heap}_{\gamma'_h}(h'_1)$ and $\mathrm{region}(r, \gamma_h, \gamma'_h)$, then we have `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 $\mathit{heap}_{\gamma_h}(h'_2)$.

By Lemma 3.1 (3), it suffices to show: $\mid\!\Rrightarrow \exists h'_2, w'.\ \big\langle h'_1, \texttt{runST}\ \{v'\}\big\rangle \to_d^* \big\langle h'_2, w'\big\rangle * \mathit{heap}_{\gamma'_h}(h'_2) * [\![\Xi \vdash \tau]\!]_\Delta(w, w')$. 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 3.3 (Fundamental theorem). $\Xi \mid \Gamma \vdash e : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\log} e : \tau$

Theorem 3.4 (Soundness of logical relation).

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

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

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

By unfolding the IC predicate, we get the assumption that $\langle h_1, e\rangle \rightarrow^* \langle h_2, v\rangle$, 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 $\Xi \mid \Gamma \vDash e \preceq_{\log} e : \mathbf{1}$, to get that $\left\langle h'_1, e\right\rangle \rightarrow^*_d \left\langle h'_2, v'\right\rangle$ 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 $\Xi \mid \Gamma \vDash e \preceq_{\log} e : \mathbf{1}$, 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, $\left\langle h'_1, ()\right\rangle \rightarrow^*_d \left\langle h'_1, ()\right\rangle$ and that $(v, ())$ are related at the unit type. □

Theorem 4.2 (Commutativity). *If* $\Xi \mid \Gamma \vdash e_1 : \tau_1$ *and* $\Xi \mid \Gamma \vdash e_2 : \tau_2$ *then*

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

Proof Sketch. We only show $\Xi \mid \Gamma \vDash \texttt{let}\, x = e_2\, \texttt{in}\, (e_1, x) \;\preceq_{\log}\; (e_1, e_2) \;:\; \tau_1 \times \tau_2$, the other direction is similar. Unfolding the IC predicate we get the assumption that $\langle h_1, \texttt{let}\, x = e_2\, \texttt{in}\, (e_1, x)\rangle \rightarrow^* \langle h_2, v\rangle$ 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 $\left\langle h'_1, (e_1, e_2)\right\rangle \rightarrow^*_d \left\langle h'_2, v'\right\rangle$ 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 $\langle h_1, e_2\rangle \rightarrow^* \langle h_3, v_2\rangle$, $\langle h_3, e_1\rangle \rightarrow^* \langle h_2, v_1\rangle$ 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 $\Xi \mid \Gamma \vDash e_1 \preceq_{\log} e_1 : \tau_1$ (which follows from the fundamental theorem) to conclude[6] $\left\langle h'_1, e_1\right\rangle \rightarrow^*_d \left\langle h'_3, v'_1\right\rangle$ 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

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

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

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'', $\mathit{heap}_{\gamma''}(h'_3)$. We use this, together with $\mathit{heap}_{\gamma_h}(h_1)$ (which we originally got by unfolding the IC predicate) and $\Xi \mid \Gamma \vDash e_2 \preceq_{\log} e_2 : \tau_2$ (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 $\mathit{heap}_{\gamma''}(h'_2)$ and $\mathit{heap}_{\gamma_h}(h_3)$ 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 $\Xi \mid \Gamma \vDash e_1 \preceq_{\log} e_1 : \tau_1$ (which we know from the fundamental theorem) together with $\mathit{heap}_{\gamma_h}(h_3)$ (from simulating e_2) and $\mathit{heap}_{\gamma'_h}(h'_1)$ (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 $\mathit{heap}_{\gamma'_h}(h''_3)$ and $\mathit{heap}_{\gamma_h}(h_2)$ 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 $\mathit{heap}_{\gamma'_h}(h_2)$ (the heap at the end of execution of the specification side) whereas we own $\mathit{heap}_{\gamma'_h}(h''_3)$ which is the heap of the specification side after execution of e_1 and before execution of e_2. However, using some resource reasoning (which depends on details explained in §5), we can conclude that $h''_3 \subseteq h_2$. This in turn allows us to update our heap resource to get $\mathit{heap}_{\gamma'_h}(h_2)$, 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 4.3 (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 (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) $\texttt{let}\, z = e_1 \,\texttt{in}\, \texttt{rec}\, f(x) = \texttt{let}\, y = e_1 \,\texttt{in}\, e_2 \preceq_{\mathrm{ctx}} \texttt{rec}\, f(x) = \texttt{let}\, y = e_1 \,\texttt{in}\, e_2 : \tau_1 \to \tau_2$ 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; $\preceq_{\log}^{NN}$. 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

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

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 $e_1 \preceq_{\log}^{NN} e_1 : \tau' \to \tau''$ 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 4.5 (η expansion for Rec). *If* $\Xi \mid \Gamma \vdash e : \tau_1 \to \tau_2$ *then* $\Xi \mid \Gamma \vDash e \preceq_{\mathrm{ctx}} \texttt{rec}\, f(x) = e\ x : \tau_1 \to \tau_2$

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

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

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 4.6 (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*

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

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 $e_1 = \texttt{inj}_i\, e$. The induction hypothesis tells us that $\Xi \mid \Gamma \vDash (\lambda x.\, e)\ e_2 \preceq_{\mathrm{ctx}} e[e_2/x] : \tau_i$ and we have to show that $\Xi \mid \Gamma \vDash (\lambda x.\, \texttt{inj}_i\, e)\ e_2 \preceq_{\mathrm{ctx}} (\texttt{inj}_i\, e)[e_2/x] : \tau_1 + \tau_2$. Notice that it is easy to prove (using the fundamental theorem) that $\Xi \mid \Gamma \vDash (\lambda x.\, \texttt{inj}_i\, e)\ e_2 \preceq_{\log} \texttt{inj}_i\, ((\lambda x.\, e)\ e_2) : \tau_1 + \tau_2$ 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 $\Xi \mid \Gamma, y : \tau_1, f : \tau_1 \to \tau_2 \vDash (\lambda x.\, e)\ e_2 \preceq_{\mathrm{ctx}} e[e_2/x] : \tau_2$ and we have to show that $\Xi \mid \Gamma \vDash (\lambda x.\, (\mathtt{rec}\, f(y) = e))\ e_2 \preceq_{\mathrm{ctx}} (\mathtt{rec}\, f(y) = e)[e_2/x] : \tau_1 \to \tau_2$ or equivalently (by simply massaging the terms) $\Xi \mid \Gamma \vDash \mathtt{let}\, x = e_2\, \mathtt{in}\, (\mathtt{rec}\, f(y) = e) \preceq_{\mathrm{ctx}} (\mathtt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$. By rec Hoisting and transitivity of contextual refinement, it suffices to show $\Xi \mid \Gamma \vDash (\mathtt{rec}\, f(y) = \mathtt{let}\, x = e_2\, \mathtt{in}\, e) \preceq_{\mathrm{ctx}} (\mathtt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$ 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

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

which is a consequence of the corresponding logical relatedness.

Theorem 4.7 (Equations for stateful computations). *See Figure 3.*

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

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

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:

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

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. □

5 IRIS DEFINITIONS OF PREDICATES USED IN THE LOGICAL RELATION

In this section we detail how the abstract predicates (regions, $\mathrm{region}(r, \gamma_h, \gamma_h')$, $\mathrm{isRgn}(\alpha, r)$, $\mathit{heap}_{\gamma_h}(h)$ 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.

5.1 Invariants, Saved Predicates and Ghost State

We extend the grammar for Iris propositions P, presented in §3 with syntax for invariants, saved predicates and ghost-resources:

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

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]:

$$\textsc{Inv-alloc}\quad \frac{P}{\mathrel{|\!\!\Rrightarrow} \boxed{P}} \qquad\qquad \textsc{Inv-open}\quad \frac{P \Rrightarrow\!\!\ast\, P \ast Q}{\boxed{P} \Rrightarrow\!\!\ast\, Q}$$

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, $\boxed{P} \dashv\vdash \boxed{P} \ast \boxed{P}$.

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).

$$\textsc{SavedPred-Alloc}\quad \mathrel{|\!\!\Rrightarrow}_{\mathcal{E}} \exists \gamma.\, \gamma \Mapsto \Phi \qquad \textsc{SavedPred-Persistent}\quad \gamma \Mapsto \Phi \dashv\vdash \gamma \Mapsto \Phi \ast \gamma \Mapsto \Phi \qquad \textsc{SavedPred-Equiv}\quad \frac{\gamma \Mapsto \Phi \ast \gamma \Mapsto \Psi}{\triangleright \Phi(a) \vdash \triangleright \Psi(a)}$$

The later modality is used in rule SavedPred-Equiv as a guard to avoid self referential paradoxes [Krebbers et al. 2017a], which is not so surprising, after all, since saved propositions essentially allow us to store a predicate (something of type $\kappa \longrightarrow \mathit{iProp}$) 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 $\checkmark : \mathcal{M} \longrightarrow \mathit{iProp}$, 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.

$$\textsc{Ghost-Alloc}\quad \checkmark a \vdash \mathrel{|\!\!\Rrightarrow} \exists \gamma.\, \boxed{a}^{\gamma} \qquad \textsc{Own-Valid}\quad \boxed{a}^{\gamma} \vdash \checkmark(a) \qquad \textsc{Sharing}\quad \boxed{a}^{\gamma} \ast \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$$

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 [Krebbers et al. 2017a]), but focus instead on the properties which the elements of the monoids satisfy, shown in Figure 8. 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.

[7] Technically,$\Rrightarrow$ has masks $\Rrightarrow_{\mathcal{E}}$ 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.

[8] The rules hold for invariants $\boxed{P}$ where P is *timeless*. For details see [Krebbers et al. 2017a].

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
$\not\checkmark(\mathrm{ex}(a) \cdot b)$

Frag-distributes
$\circ\, a \cdot \circ\, b = \circ\,(a \cdot b)$

Full-Exclusive
$\not\checkmark(\bullet\, a \cdot \bullet\, b)$

Auth-Alloc-Finset
$$\frac{h \cap a = \emptyset}{\boxed{\bullet\, h}^{\gamma} \Rrightarrow\!\!\ast \boxed{\bullet\,(h \uplus a) \cdot \circ\, a}^{\gamma}}$$

Auth-Alloc-Fpfn
$$\frac{\mathrm{dom}(h) \cap \mathrm{dom}(a) = \emptyset}{\boxed{\bullet\, h}^{\gamma} \Rrightarrow\!\!\ast \boxed{\bullet\,(h \uplus a) \cdot \circ\, a}^{\gamma}}$$

Agree
$\mathrm{ag}(a) \cdot \mathrm{ag}(a) = \mathrm{ag}(a)$

Fpfn-operation-success
$$(a \cdot b)(x) = \begin{cases} a(x) & \text{if } x \in \mathrm{dom}(a) \land x \notin \mathrm{dom}(b) \\ a(x) \cdot b(x) & \text{if } x \in \mathrm{dom}(a) \cap \mathrm{dom}(b) \\ b(x) & \text{if } x \in \mathrm{dom}(b) \land x \notin \mathrm{dom}(a) \end{cases}$$

Auth-Update-Fpfn
$$\boxed{\bullet\,(h \uplus (\ell \mapsto \mathrm{ex}(v_1))) \cdot \circ\, \ell \mapsto \mathrm{ex}(v_1)}^{\gamma} \Rrightarrow\!\!\ast \boxed{\bullet\,(h \uplus (\ell \mapsto \mathrm{ex}(v_2))) \cdot \circ\, \ell \mapsto \mathrm{ex}(v_2)}^{\gamma}$$

Fig. 8. Rules for selected monoid resources in Iris

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

$$\textsc{Heap} \triangleq \textsc{Auth}(Loc \xrightarrow{\mathrm{fin}} (\mathrm{Ex}(Val))) \qquad heap_{\gamma}(h) \triangleq \boxed{\bullet\, h}^{\gamma} \qquad \ell \mapsto_{\gamma} v \triangleq \boxed{\circ\,[l \mapsto \mathrm{ex}(v)]}^{\gamma}$$

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.

5.2 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:

$$\mathsf{Rel} \triangleq \textsc{Auth}((Loc \times Loc) \xrightarrow{\mathrm{fin}} (\textsc{Ag}(Names))) \qquad \mathsf{Bij} \triangleq \textsc{Auth}(\mathcal{P}(Loc \times Loc))$$

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:

$$\mathsf{Region} \triangleq \textsc{Auth}(\mathbb{Z}^{+} \xrightarrow{\text{fin}} (\textsc{Ag}(\mathit{Names} \times \mathit{Names})))$$

We fix a global ghost name γ_{reg} for an instance of this last monoid. For Region, ownership of $\boxed{\circ\, r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ 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 $\boxed{\circ\, r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ 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,

$$\boxed{\circ\, r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ\, r \mapsto \mathrm{ag}(\gamma'_{bij}, \gamma'_{rel})}^{\gamma_{reg}} \vdash \gamma_{bij} = \gamma'_{bij} \wedge \gamma_{rel} = \gamma'_{rel} \tag{6}$$

We can now present the $\mathsf{region}(r, \gamma_h, \gamma'_h)$ predicate in detail:

$$\begin{aligned}\mathsf{region}(r, \gamma_h, \gamma'_h) \triangleq \exists R, \gamma_{bij}, \gamma_{rel}.\ & \boxed{\circ\, r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel}) : \mathsf{Region}}^{\gamma_{reg}} * \boxed{\bullet\, R : \mathsf{Rel}}^{\gamma_{rel}} * \\ & \mathop{\Large\ast}_{(\ell,\ell') \mapsto \mathrm{ag}(\gamma_{pred}) \in R} \left(\begin{array}{l} \exists \Phi : (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}), v, v'.\ \ell \mapsto_{\gamma_h} v\ * \\ \qquad \ell' \mapsto_{\gamma'_h} v' * \gamma_{pred} \Mapsto \Phi * \triangleright \Phi(v, v') \end{array} \right)\end{aligned}$$

The predicate asserts that the semantic region r is associated with two ghost names, γ_{bij} and γ_{rel}, by $\boxed{\circ\, r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, and full authoritative ownership of R, which is a mapping of pairs of locations to ghost names. Further, for each element $(\ell, \ell') \mapsto \mathrm{ag}(\gamma_{pred}) \in R$ 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 $\boxed{\bullet\, M : \mathsf{Reg}}^{\gamma_{reg}}$:

$$\mathsf{regions} \triangleq \boxed{\begin{array}{l} \exists M.\ \boxed{\bullet\, M : \mathsf{Region}}^{\gamma_{reg}}\ * \\ \mathop{\Large\ast}_{r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel}) \in M} \left(\begin{array}{l} \exists g : \mathsf{finset}(\mathit{Loc} \times \mathit{Loc}), R : (\mathit{Loc} \times \mathit{Loc}) \xrightarrow{\text{fin}} (\textsc{Ag}(\mathit{Names})). \\ \boxed{\bullet\, g}^{\gamma_{bij}} * \mathsf{bijection}(g) * \boxed{\circ\, R}^{\gamma_{rel}} * g = \mathrm{dom}(R) \end{array} \right) \end{array}}$$

For each element $r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})$ 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 $\boxed{\circ\, r \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel}) : \mathsf{Region}}^{\gamma_{reg}}$ 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 `STRef` $\rho\ \tau$[9]:

$$\mathsf{isRgn}(\alpha, r) \triangleq \exists \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ \alpha = r * \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$$
$$\mathsf{bij}(r, \ell, \ell') \triangleq \exists \gamma_{bij}, \gamma_{rel}.\ \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ\, (\ell, \ell')}^{\gamma_{bij}}$$
$$\mathsf{rel}(r, \ell, \ell', \Phi) \triangleq \exists \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ \boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ\, [(\ell, \ell') \mapsto \mathsf{ag}(\gamma_{pred})]}^{\gamma_{bij}} * \gamma_{pred} \Mapsto \Phi$$

Each of the predicates owns the ghost resource suggested by its name. For instance, Property (5) from Section 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 $r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})$, to obtain $\boxed{\circ\, r \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, for some r not in $\mathsf{dom}(M)$, 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 $\mathsf{dom}(\emptyset) = \emptyset$ hold trivially, so we have reestablished the body of the invariant.

6 FORMALIZATION IN COQ

We have formalized our technical development and proofs in the Iris implementation in Coq [Krebbers et al. 2017a,b]. The Iris implementation in Coq [Krebbers et al. 2017a] includes a model of Iris and proof of soundness of the Iris logic itself. The Iris Proof Mode (IPM) [Krebbers et al. 2017b] 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 [Krebbers et al. 2017a] 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 plain 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

[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.

[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

data structure in Coq that represents the language. 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, ■ 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 `%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 [Krebbers et al. 2017b], 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 [Schäfer et al. 2015]. 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]

7 RELATED WORK

The most closely related work is the original seminal work of Launchbury and Peyton Jones [1994], which we discussed and related to in the Introduction. In this section we discuss other related work.

Moggi and Sabry [2001] 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)*" [Moggi and Sabry 2001].

It was pointed out already in [Launchbury and Peyton Jones 1994] that there seems to be a connection between encapsulation using runST and effect masking in type-and-effect systems à la Gifford and Lucassen [1986]. This connection was formalized by Semmelroth and Sabry [1999], 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

[11] About 17 minutes on a laptop using "make -j4" to compile our Coq formalization of about 12,500 lines.

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 [Benton and Buchlovsky 2007; Benton et al. 2007, 2009, 2006] and their work was extended by Thamsborg and Birkedal [2011] 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 [Thamsborg and Birkedal 2011] 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 [Thamsborg and Birkedal 2011]. 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 [Thamsborg and Birkedal 2011], 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 [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017]. 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.

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. In the original paper [Launchbury and Peyton Jones 1994], 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 [Krogh-Jespersen et al. 2017] would hold for such a language.

ACKNOWLEDGMENTS

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP*.

Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References Embeddable in Higher-Order Logic. In *Proceedings of 17th Annual IEEE Symposium Logic in Computer Science*. IEEE Computer Society Press, 75–86.

Andrew Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* 23, 5 (2001), 657–683.

Andrew Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *POPL*.

Nick Benton and Peter Buchlovsky. 2007. Semantics of an effect analysis for exceptions. In *TLDI*.

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2007. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*.

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational semantics for effect-based program transformations: higher-order store. In *PPDP*.

Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. 2006. Reading, writing and relations. In *PLAS*. Springer.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-Indexed Kripke Models over Recursive Worlds. In *POPL*.

D. Dreyer, A. Ahmed, and L. Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (2011).

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *TCS* 103, 2 (1992), 235–271.

D. K. Gifford and J. M. Lucassen. 1986. Integrating functional and imperative programming. In *LISP*.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650.

Robbert Krebbers, Ralf Jung, AleÅą Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *European Symposium on Programming (ESOP)*.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*.

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *POPL*.

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 24–35.

John Launchbury and Simon L. Peyton Jones. 1995. State in haskell. *Lisp and symbolic computation* 8, 4 (1995), 293–341.

E. Moggi and Amr Sabry. 2001. Monadic Encapsulation of Effects: A Revised Approach (Extended Version). *J. Funct. Program.* 11, 6 (Nov. 2001), 591–627.

John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. *Information Processing* (1983).

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *ITP (LNCS)*, Vol. 9236. 359–374.

Miley Semmelroth and Amr Sabry. 1999. Monadic Encapsulation in ML. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*. ACM, New York, NY, USA, 8–17.

Jacob Thamsborg and Lars Birkedal. 2011. A Kripke logical relation for effect-based program transformations. In *ICFP*.