A Logical Relation for Monadic Encapsulation of State
Proving contextual equivalences in the presence of runST

AMIN TIMANY, imec-Distrinet, KU-Leuven
LÉO STEFANESCO, ENS Lyon, Université de Lyon
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 contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of stateful computations encapsulated using runST.

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. *PACM Progr. Lang.* 1, 1, Article 1 (January 2017), 24 pages.
DOI: 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 in 1994 (Launchbury and Peyton Jones 1994). The ST monad comes equipped with a function $\mathsf{runST} : (\forall \beta, \mathsf{ST}\ \beta\ \tau) \to \tau$ 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 higher-order functional programming language with impredicative polymorphism, recursive types, and a Haskell-style ST monad type with runST. In contrast to earlier work, we give an operational semantics of STLang with a 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. This means that compilers can use these refinements and equivalences to optimize code.

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. Here 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

2017. 2475-1421/2017/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

certain modalities which we use to simplify the construction of the logical relations. Logical relations for other type systems have been defined in Iris before (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.

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 Kleisly 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, as is usual, the type of the contents of the reference cells, τ, but also another type parameter, β, which, intuitively, indicates which (logical) region of the heap this reference belongs to. The interesting part of the interface is the interaction of this type parameter with the following function

```
runST :: (∀ β. ST β α) → α
```

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

Finally, equality on references is decidable:

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

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

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 → ST β Integer → ST β Integer → ST β Integer
```

Hence, the argument of `runST` has type `(∀ β. ST β Integer)` 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 (Launchbury and Peyton Jones

[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.

```haskell
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 : ℕ -> ℕ =
  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.

1994) is that the parametric polymorphism in `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, then 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, and thus it is safe to deallocate 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

$$\texttt{let}\, x = e\, \texttt{in}\, (x, x) \quad \approx_{\text{ctx}} \quad (e, e) \tag{1}$$

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 (1) 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.

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 the challenge of showing that `runST` can be used to encapsulate state properly. Earlier work has focused mostly 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 global mutable heap. In particular, the semantics and parametricity results of Launchbury and Peyton-Jones (Launchbury and Peyton Jones 1994) is denotational and does not use a global mutable heap with in-place update, and they state (Launchbury and Peyton Jones 1994, 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 §6.

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 $\Xi \mid \Gamma \vDash e \preceq_{\text{ctx}} e' : \tau$, in much the standard way. As usual, e and e' are contextually equivalent, denoted $\Xi \mid \Gamma \vDash e \approx_{\text{ctx}} e' : \tau$, if e contextually refines e' and vice versa. 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. In particular, a compiler can use them to optimize code.

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 §5.

Refinement (Neutrality) expresses that a computation of unit type either diverges or produces the unit value. The contextual equivalence in (Commutativity) expresses that the order of evaluation for pure computations

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

Fig. 2. Contextual Refinements and Equivalences for Pure Computations

does not matter: the computation on the left first evaluates e_2 and then e_1, on the right we first evaluate e_1 and then e_2. The contextual equivalence in (Idempotency) expresses the idempotency of pure computations: it does not matter whether we evaluate an expression once, as done on the left, or twice, as done on the right. The contextual

refinements in (Rec hoisting) and (Λ hoisting) express the soundness of λ-hoisting for ordinary recursive functions and for type functions. (Note that the other direction of refinement does not hold, since the hoisted expression may diverge.) 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.

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

Fig. 3. Contextual Equivalences for Stateful Computations

The results in Figure 2 are the kind of results one would expect for pure computations; the challenge is, of course, to show that they hold in the full STLang language, that is, also when subexpressions may involve arbitrary (possibly nested) stateful computations encapsulated using `runST`. That is the purpose of our logical relation, which we present in §4. Since the logical relation is defined using the Iris logic, we first present some background information on Iris in §3. The logical relation is defined over the operational semantics and provides an interpretation of the types of STLang. As usual for call-by-value languages, we give both a value and an expression interpretation of types. The value interpretation of the pure types is fairly standard. The expression interpretation and the value interpretation of the types `STRef` $\rho\ \tau$ and ST $\rho\ \tau$ relate the concrete heap used in the operational semantics to an abstraction which, loosely speaking, captures how we can think of the heap as being divided up into disjoint regions. In §5 we show how the logical relation can be used to prove the contextual equivalences for pure computations listed in Figure 2. We discuss related work in §6 and conclude in §7.

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 that contextual refinements and equivalences that are expected to hold for pure computations do indeed hold in the presence of stateful computations. This is the first time that these results have been established for a programming language with an operational semantics that uses a global higher-order heap with in-place destructive updates.
- We formalize our logical relation in the logic of 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 §3.

2 THE STLang LANGUAGE

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

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 `ref(e)` creates a new reference, `!e` reads from one and `e` ← `e` writes into one. Finally, `runST` runs effectful

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

Fig. 4. The syntax of STLang.

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

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

$$
\textsc{Tinst}\ \frac{\Xi \mid \Gamma \vdash e : \forall X.\, \tau \qquad \Xi \vdash \tau'}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/X]} \qquad
\textsc{Tnew}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \vdash \rho}{\Xi \mid \Gamma \vdash \texttt{ref}(e) : \mathsf{ST}\ \rho\ (\mathsf{STRef}\ \rho\ \tau)} \qquad
\textsc{Tderef}\ \frac{\Xi \mid \Gamma \vdash e : \mathsf{STRef}\ \rho\ \tau}{\Xi \mid \Gamma \vdash\ !\, e : \mathsf{ST}\ \rho\ \tau}
$$

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

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

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

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

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 operational semantics for STLang, using a transition system $\langle h, e\rangle \to \langle h', e'\rangle$ whose nodes are configurations consisting of a heap h and an expression e. A heap $h \in \mathit{Loc} \rightharpoonup^{\text{fin}} \mathit{Val}$ is a finite partial function that associates values to locations, which we suppose are natural numbers ($\mathit{Loc} \triangleq \mathbb{N}$).

The semantics is shown in Figure 6. The semantics 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

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

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

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

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

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, $\langle [l \mapsto 3], \mathtt{bind}\,(\mathtt{return}\, l)\,\mathtt{in}\,(\lambda\, r.\ \cdots)\rangle \rightsquigarrow_h \langle [l \mapsto 3], (\lambda\, r.\ \cdots)\, l\rangle$ to get:

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

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

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

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

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

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

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

Finally we get:

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

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 defining rules for this relation are standard and can be found in the accompanying (Technical Report 2017).

Definition 2.1 (Contextual refinement).

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

where

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

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:

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

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

$$\langle h, (\mathtt{rec}\, f(x) = e)\ v\rangle \to_h \langle h, e[v, \mathtt{rec}\, f(x) = e/x, f]\rangle \qquad \langle h, (\Lambda\, e)\ \_\rangle \to_h \langle h, e\rangle$$

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

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

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 CBV small-step operational semantics.

Notice that the definition of contextual refinement above does not assume that both programs start in the empty heap or appropriately related heaps as is usual. This can be thought of as an aspect of purity: regardless of the heaps, if the $C[e]$ reduces to a value so does $C[e']$.

Definition 2.2 (Contextual equivalence).

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

3 IRIS

Iris was originally presented as a framework for higher-order (concurrent) separation logic, with built-in notions of 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).

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). 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 *loc. cit.* were defined using Iris's built-in notion of Hoare triple / weakest precondition. It turns out that this approach to representing logical relations is 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 paper we therefore use the Iris base logic to define a couple of new logical connectives which we use, instead of weakest preconditions, to define our binary logical relation. In this section we explain the bits we need and define the new logical connectives that we use in the next section to define our logical relation. We have to omit some details, which can be found in (Krebbers et al. 2017a).

3.1 Iris logic

The Iris logic is a higher-order logic, in which one can quantify over the Iris types κ:

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

Here *Expr* and *Val* are the types of syntactic expressions and values of STLang, $\kappa \xrightarrow{\text{fin}} \kappa$ is the type of finite functions, $\mathsf{finset}(\kappa)$ is the type of finite sets, $\mathbb{N}$ is the type of natural numbers, $\mathbb{B}$ is the type of booleans, *Names* is a type of ghost names, *Monoid* is a type of monoids, and *iProp* is the type of Iris propositions.

The grammar for Iris propositions P

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rrightarrow P \mid P \vee P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \mid$$
$$\triangleright P \mid \mu r.P \mid \checkmark(a) \mid \Box P \mid \rtimes P \mid \Diamond P \mid \Rrightarrow_{\mathcal{E}} P \mid \boxed{M}^{\gamma} \mid \boxed{P} \mid \gamma \Mapsto \Phi \mid \ldots$$

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

The $\triangleright$ is a modality, called "later", which is used to guard recursively defined propositions: $\mu r.P$ is a well-defined guarded-recursive predicate 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 that we use is the "always" modality ($\Box$). Intuitively, $\Box P$ holds whenever P holds and is a duplicable assertion. In particular we have $(\Box P) * (\Box P) \dashv\vdash \Box P$ where $\dashv\vdash$ is the logical equivalence of formulas. We say that P is *persistent* if $P \vdash \Box P$. Persistent propositions are thus duplicable. The always modality is idempotent, $\Box P \vdash \Box\Box P$, and also satisfies $\Box P \vdash P$. The always modality (and by extension persistence) also commutes with all of the connectives of higher-order separation logic, including quantifiers.

The "naught" modality ($\rtimes$) is very similar to the always modality. Intuitively, $\rtimes P$ holds whenever P holds and asserts no ownership. In particular, if $P \vdash \rtimes Q$ then $P \vdash P * \rtimes Q$. We say that P is *plain* if $P \vdash \rtimes P$. Plain properties are those that assert no ownership of resources. A prime example here is reduciblity: $\mathsf{plain}(\langle h_1, e_1\rangle \to^* \langle h_2, e_2\rangle)$. The naught modality is idempotent, $\rtimes P \vdash \rtimes \rtimes P$, and we have $\rtimes P \vdash P$. The naught modality (and plainness by extension) also commutes with all of the connectives of the higher-order separation logic, including quantifiers, and also $\triangleright$, $\Box$ and $\Diamond$ (see below) modalities.

The "except zero" modality ($\Diamond$) is another modality of Iris that we use. Intuitively $\Diamond P$ holds if P holds for any step *but* the zeroth. The except zero modality is idempotent, $\Diamond\Diamond P \vdash \Diamond P$, and also satisfies $P \vdash \Diamond P$.

Another important modality in Iris is the "update" modality[2] ($\Rrightarrow_{\mathcal{E}}$). The mask, $\mathcal{E} \in \{\top, \bot\}$, in $\Rrightarrow_{\mathcal{E}}$ is to make sure that invariants (see below) are now opened in a nested fashion.[3] We drop the mask when it is $\top$ which indicates no invariant is open. Intuitively, the proposition $\Rrightarrow_{\mathcal{E}} P$ holds for resources that can be updated (allocated, deallocated, or changed), by opening invariants if necessary (and allowed by the mask), to resources that satisfy P, without violating the environment's knowledge or ownership of resources. We write $P \Rrightarrow_{\mathcal{E}} Q$ as a shorthand for $P \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} Q$. The update modality is idempotent, $\Rrightarrow_{\mathcal{E}} (\Rrightarrow_{\mathcal{E}} P) \dashv\vdash \Rrightarrow_{\mathcal{E}} P$.

We have the following properties for Iris modalities:

$$\textsc{Except-zero-intro}\quad P \vdash \Diamond P \qquad \textsc{Later-intro}\quad P \vdash \triangleright P \qquad \textsc{Update-intro}\quad P \vdash \Rrightarrow P \qquad \textsc{Mod-mono}\quad \frac{P \vdash Q}{\bowtie P \vdash \bowtie Q}$$

where $\bowtie$ is any modality ($\triangleright$, $\Box$, $\rtimes$, $\Diamond$, or $\Rrightarrow_{\mathcal{E}}$).

One of the most interesting aspects of plain propositions is how they interact with the update modality.

Lemma 3.1 (Update modality and plainness).

$$\textit{If } P \textit{ is plain and } Q \vdash P \textit{ then } \Rrightarrow_{\mathcal{E}} Q \vdash \Rrightarrow_{\mathcal{E}} ((\Rrightarrow_{\mathcal{E}} Q) * P)$$

This lemma basically says that if we can update resources to have Q which implies a plain proposition P then we can have P without actually having to perform the whole updating of the resources.

Iris also features invariants, $\boxed{P}$, which are typically used to enforce that a proposition P holds for some state shared among several threads. In this paper we will use certain simple kinds of invariants and therefore we can use the following rules for allocating and opening invariants:

$$\textsc{Inv-alloc}\quad P \Rrightarrow_{\mathcal{E}} \boxed{P} \qquad \textsc{Inv-open}\quad \frac{P \Rrightarrow_{\bot} P * Q}{\boxed{P} \Rrightarrow_{\top} Q}$$

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 special kind of invariants that we use in this paper.[4] Invariants are persistent, $\boxed{P} \dashv\vdash \boxed{P} * \boxed{P}$.

Resources in Iris are described using a kind of partial commutative monoids, and the user of the logic can introduce new monoids. The partiality comes from the fact that disjoint union of finite maps is partial. Undefinedness

[2] In (Krebbers et al. 2017a) this modality is called the *fancy* update modality.

[3] In Iris invariants are named and these masks in general range over the set of invariant names and allow more fine-grained control over how invariants are opened. Here we only present this simpler version of invariants.

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

$$\textsc{Ghost-Alloc}\ \frac{\checkmark a}{\Rrightarrow_{\mathcal{E}} \exists \gamma.\ \boxed{a}^{\gamma}} \qquad \textsc{Own-Valid}\ \boxed{a}^{\gamma} \vdash \checkmark(a) \qquad \textsc{Sharing}\ \boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$$

Fig. 7. Rules for ghost resources in Iris

$$\textsc{Agree}\ \mathrm{ag}(a) \cdot \mathrm{ag}(a) = \mathrm{ag}(a) \qquad \textsc{Agreement-Valid}\ \checkmark(\mathrm{ag}(a) \cdot \mathrm{ag}(b)) \dashv\vdash a = b \qquad \textsc{Exclusive}\ \not\checkmark(\mathrm{ex}(a) \cdot b) \qquad \textsc{Auth-Included}\ \bullet a \cdot \circ b \vdash a \subseteq b \qquad \textsc{Frag-distributes}\ \circ a \cdot \circ b = \circ(a \cdot b)$$

$$\textsc{Full-Exclusive}\ \not\checkmark(\bullet a \cdot \bullet b) \qquad \textsc{Auth-Alloc-Finset}\ \frac{h \cap a = \emptyset}{\boxed{\bullet h}^{\gamma} \Rrightarrow_{\mathcal{E}} \boxed{\bullet(h \uplus a) \cdot \circ a}^{\gamma}} \qquad \textsc{Auth-Alloc-Fpfn}\ \frac{\mathrm{dom}(h) \cap \mathrm{dom}(a) = \emptyset}{\boxed{\bullet h}^{\gamma} \Rrightarrow_{\mathcal{E}} \boxed{\bullet(h \uplus a) \cdot \circ a}^{\gamma}}$$

$$\textsc{Fpfn-valid}\ \checkmark(a) \dashv\vdash \forall x \in \mathrm{dom}(a).\ \checkmark(a(x)) \qquad \textsc{Fpfn-operation-success}\ a \cdot b = \begin{cases} a(x) & \text{if } x \in \mathrm{dom}(a) \wedge x \notin \mathrm{dom}(b) \\ a(x) \cdot b(x) & \text{if } x \in \mathrm{dom}(a) \cap \mathrm{dom}(b) \\ b(x) & \text{if } x \in \mathrm{dom}(b) \wedge x \notin \mathrm{dom}(a) \end{cases}$$

$$\textsc{Auth-update-Fpfn}\ \boxed{\bullet(h \uplus (\ell \mapsto \mathrm{ex}(v_1))) \cdot \circ \ell \mapsto \mathrm{ex}(v_1)}^{\gamma} \Rrightarrow_{\mathcal{E}} \boxed{\bullet(h \uplus (\ell \mapsto \mathrm{ex}(v_2))) \cdot \circ \ell \mapsto \mathrm{ex}(v_2)}^{\gamma}$$

Fig. 8. Rules for selected monoid resources in Iris

is treated by means of a validity predicate $\checkmark : \mathcal{M} \to \mathit{iProp}$, which, for a monoid $\mathcal{M}$, expresses which elements are valid / defined.

We write $\boxed{M}^{\gamma}$ to assert that a monoid instance named γ has contents M. We think of this assertion as a ghost variable γ with contents M. The relevant rules regarding ghost ownership are depicted in Figure 7.

This figure shows the rule Alloc used for allocating new instances of monoids (resources). It intuitively says that we can always allocate any monoid element a so long as it is *valid* ($\checkmark a$).

The last kind of Iris proposition that we use is called saved propositions $\gamma \Mapsto \Phi$. This is simply a mechanism for 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).

$$\textsc{SavedPred-Alloc}\ \Rrightarrow_{\mathcal{E}} \exists \gamma.\ \gamma \Mapsto \Phi \qquad \textsc{SavedPred-Persistent}\ \gamma \Mapsto \Phi \dashv\vdash \gamma \Mapsto \Phi * \gamma \Mapsto \Phi \qquad \textsc{SavedPred-Equiv}\ \frac{\gamma \Mapsto \Phi * \gamma \Mapsto \Psi}{\triangleright \Phi(a) \vdash \triangleright \Psi(a)}$$

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

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. It also depicts the rules necessary for allocating and updating finite set, $\mathsf{finset}(A)$, and finite partial function, $A \rightharpoonup^{\mathrm{fin}} M$,

monoids. In these monoids, the monoid operation, $\cdot$, is *disjoint* union. The notation $a \mapsto b : A \rightharpoonup^{\text{fin}} B \triangleq \{(a, b)\}$ is a singleton finite partial function. Notice that these rules are stated only for monoids that we use in this work and not in Iris in its generality. For instance in the rule Auth-Included $\subseteq$ is a set relation and is defined for finite set and finite partial function monoids and not in general. The 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 $\text{Ex}(M)$. As the name suggests, the operation of the agreement monoid guarantees that $\text{ag}(a) \cdot \text{ag}(b)$ 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.

$$\boxed{\text{ag}(a)}^{\gamma} \dashv\vdash \boxed{\text{ag}(a) \cdot \text{ag}(a)}^{\gamma} \dashv\vdash \boxed{\text{ag}(a)}^{\gamma} * \boxed{\text{ag}(a)}^{\gamma}$$

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.

Notice that the monoids can be nested. For instance, the update rule for finite partial functions (Auth-update-Fpfn) is defined for a monoid of the form $\textsc{Auth}(A \rightharpoonup^{\text{fin}} \text{Ex}(M))$. From this rule we can easily show why the specialized update rule (Auth-update-Fpfn) holds.

3.2 Future modality and If-convergence

In this subsection, we define two new constructs in Iris, which we will use in our definition of the logical relation in the following section. 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* $\models\!\!\!|\{\cdot\}\!\!\!\models\!\!\!\triangleright$ as follows:

$$\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright P \triangleq (\models\!\!\!\triangleright \triangleright)^n \models\!\!\!\triangleright P$$

where $(\models\!\!\!\triangleright \triangleright)^n$ is n times repetition of $\models\!\!\!\triangleright \triangleright$. Notice that the mask on the update modality is $\top$ and therefore omitted. Intuitively, $\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright P$ expresses that n steps into the future, we can update our resources to satisfy P.

Similar to the update modality, we write $P \equiv\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q$ as a shorthand for $P \mathrel{-\!\!*} \models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q$.

Lemma 3.2 (Properties of the future modality). *The future modality, $\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright$, has the following properties:*

(1) $Q \vdash \models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q$
(2) If $P \vdash Q$ *then* $\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright P \vdash \models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q$
(3) $\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q \dashv\vdash \models\!\!\!\triangleright(\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q)$
(4) $\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q \dashv\vdash \models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright(\models\!\!\!\triangleright Q)$
(5) $(\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q) * (\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q') \vdash \models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright(Q * Q')$
(6) $\triangleright^n \diamond P \vdash \models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright P$
(7) If $(\models\!\!\!|\{m - n\}\!\!\!\models\!\!\!\triangleright P) * Q \vdash Q'$ *then* $(\models\!\!\!|\{m\}\!\!\!\models\!\!\!\triangleright P) * (\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q) \vdash (\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q')$
(8) If P *is plain and* $Q \vdash P$ *then* $(\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q) \vdash (\models\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright Q) * \triangleright^n \diamond P$

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

$$\text{IC}^{\gamma}\ e\ \{\!|v.\ Q|\!\} \triangleq \forall h_1, h_2, v, n.\ \langle h_1, e\rangle \rightarrow^n \langle h_2, v\rangle * \boxed{\bullet \text{excl}(h_1)}^{\gamma} \equiv\!\!\!|\{n\}\!\!\!\models\!\!\!\triangleright \boxed{\bullet \text{excl}(h_2)}^{\gamma} * Q$$

Here v is bound in Q.[5] The function excl is a mapping from the set of heaps in STLang to the monoid $(Loc \xrightarrow{\text{fin}} \mathrm{Ex}(Val))$, which we use in the logic to represent the heap:

$$\mathrm{excl}(h) = \{(\ell, \mathrm{ex}(v)) \mid (\ell, v) \in h\}$$

The $\mathsf{IC}^\gamma\ e\ \{\!|v.\ Q|\!\}$ predicate expresses that if e with an arbitrary heap h_1 reduces in n steps to a value v and another heap h_2 and we, moreover, have ownership over the heap h_1, then n steps into the future, we will have ownership over the heap h_2 and the postcondition Q will hold. The ghost name γ is used to keep track of the contents of the heap and using that we can abstract away from the concrete heaps in when reasoning about the IC predicates.[6]

Note that the IC predicate does not require that it is *safe* to execute the expression e: if e gets stuck (or diverges) in some heap, then $\mathsf{IC}^\gamma\ e\ \{\!|v.\ Q|\!\}$ holds trivially.

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

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

The following lemma expresses useful properties of the IC predicate:

Lemma 3.3 (Properties of IC). *The IC predicate satisfies the following properties:*[7]

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

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

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

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

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

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

(7) $\triangleright(\forall \ell.\ \boxed{\circ\, \ell \mapsto ex(v)}^\gamma \Rrightarrow Q\ \ell) \vdash IC^\gamma\ \mathtt{runST}\ \{\mathtt{ref}(v)\}\ \{\!|w.\ Q|\!\}$

(8) $\triangleright\boxed{\circ\, \ell \mapsto ex(v)}^\gamma * \triangleright(\boxed{\circ\, \ell \mapsto ex(v)}^\gamma \Rrightarrow Q\ v) \vdash IC^\gamma\ \mathtt{runST}\ \{!\,\ell\}\ \{\!|w.\ Q|\!\}$

(9) $\triangleright\boxed{\circ\, \ell \mapsto ex(v')}^\gamma * \triangleright(\boxed{\circ\, \ell \mapsto ex(v)}^\gamma \Rrightarrow Q\ ()) \vdash IC^\gamma\ \mathtt{runST}\ \{\ell \leftarrow v\}\ \{\!|w.\ Q|\!\}$

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

The cases (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). We remark that properties stated in this Lemma 3.3 are the only properties of IC we need to know to prove the compatibility lemmas for the logical relation in the following section. It is only for some parts of the proofs of the lemmas justifying purity that we have to unfold the definition of IC. Even then, the definition of the future modality is never unfolded: it suffices to use the properties of the future modality stated in Lemma 3.2.

As an illuminating example let us consider the following lemma which is a simplified version of one of the sides of the idempotency property (Idempotency). We include the proof of this lemmas, since it sheds light on the general form of arguments involved in proving properties that we mentioned earlier.

Lemma 3.4 (A very basic variant of idempotency).

$$\textit{If } \Box(\forall \gamma, h_1'.\ IC^\gamma\ e\ \{\!|v.\ \exists h_2', v'.\ \langle h_1', e'\rangle \to^* \langle h_2', v'\rangle|\!\})\ \textit{then}$$
$$\forall \gamma_h, h_1'.\ IC^{\gamma_h}\ e\ \{\!|v.\ \exists h_2', v_1', v_2'.\ \langle h_1', (e', e')\rangle \to^* \langle h_2', (v_1', v_2')\rangle|\!\}$$

[5] In general the number steps, n, can also appear in Q but here we only present this slightly simpler version.

[6] This is related to the way the definition of weakest preconditions in Iris hides state (Krebbers et al. 2017a).

[7] Some of the properties listed are not the strongest provable (e.g., the case 7 could have 2 $\triangleright$ modalities), but they suffice for our purposes.

Proof. Let assume that we have $\gamma_h, h_1', h_1, h_2, v, n$ such that we own $\boxed{\bullet \mathsf{excl}(h_1)}^{\gamma_h}$ and know $\langle h_1, e\rangle \to^n \langle h_2, v\rangle$. So we have to show $\models\!\!\{n\}\!\!\Rrightarrow \boxed{\bullet \mathsf{excl}(h_2)}^{\gamma_h} * \exists h_2', v_1', v_2'.\ \langle h_1', (e', e')\rangle \to^* \langle h_2', (v_1', v_2')\rangle$. We allocate a fresh instance $\boxed{\bullet \mathsf{excl}(h_1)}^{\gamma'}$ and then duplicate and instantiate our assumption by it to get

$$\models\!\!\{n\}\!\!\Rrightarrow \boxed{\bullet \mathsf{excl}(h_2)}^{\gamma'} * \exists h_2', v'.\ \langle h_1', e'\rangle \to^* \langle h_2', v'\rangle$$

Now, using Lemma 3.2 case 8 and the fact that reductions are plain, we get $\triangleright^n \Diamond \exists h_2', v'.\ \langle h_1', e'\rangle \to^* \langle h_2', v'\rangle$. Since both later and except zero commute with quantifiers we get that there exists $h^\dagger$ and $v^\dagger$ such that $\triangleright^n \Diamond \langle h_1', e'\rangle \to^* \langle h^\dagger, v^\dagger\rangle$. By the latter fact (using Lemma 3.2 case 6) we get $\models\!\!\{n\}\!\!\Rrightarrow \langle h_1', e'\rangle \to^* \langle h^\dagger, v^\dagger\rangle$. Now we can instantiate our assumption again by $\boxed{\bullet \mathsf{excl}(h_1)}^{\gamma_h}$ and $h^\dagger$ to get $\models\!\!\{n\}\!\!\Rrightarrow \boxed{\bullet \mathsf{excl}(h_2)}^{\gamma_h} * \exists h_2', v'.\ \langle h^\dagger, e'\rangle \to^* \langle h_2', v'\rangle$.

Now we use Lemma 3.2 case 5 to get

$$\models\!\!\{n\}\!\!\Rrightarrow \langle h_1', e'\rangle \to^* \langle h^\dagger, v^\dagger\rangle * \boxed{\bullet \mathsf{excl}(h_2)}^{\gamma_h} * \exists h_2', v'.\ \langle h^\dagger, e'\rangle \to^* \langle h_2', v'\rangle$$

By monotonicity of the future modality we get that $\langle h_1', e'\rangle \to^* \langle h^\dagger, v^\dagger\rangle, \boxed{\bullet \mathsf{excl}(h_2)}^{\gamma_h}$ and that there are h_2', w' such that $\langle h^\dagger, e'\rangle \to^* \langle h_2', w'\rangle$. Finally, we have to show

$$\boxed{\bullet \mathsf{excl}(h_2)}^{\gamma_h} * \exists h_2', v_1', v_2'.\ \langle h_1', (e', e')\rangle \to^* \langle h_2', (v_1', v_2')\rangle$$

which is trivial: simply take v_1' to be $v^\dagger$ and v_2' to be w'. □

In the above proof, we had to show that the pair $\langle h_1', (e', e')\rangle$ reduces to a pair of values, and we did that by using the fact that the reduction is plain to get that there are $h^\dagger$ and $v^\dagger$ such that $\triangleright^n \Diamond \langle h_1', e'\rangle \to^* \langle h^\dagger, v^\dagger\rangle$. This was done without changing the n in the future modality $\models\!\!\{n\}\!\!\Rrightarrow \cdots$. This technique is used frequently in the proof of lemmas in §5 and we refer to it as *hypothetical* execution of e'.

4 THE LOGICAL RELATION

In this section we define a binary logical relation, which is useful for proving contextual refinement. The logical relation is shown in Figure 9 and in the following we explain the definition and present the theorems that hold for it. As mentioned in the Introduction, the value interpretation of the pure types is fairly standard, but the expression interpretation and the value interpretation of the types $\mathtt{STRef}\ \rho\ \tau$ and $\mathtt{ST}\ \rho\ \tau$ relate the concrete heap used in the operational semantics to an abstraction which captures how we can think of the heap as being divided up into disjoint regions.

The value relation $[\![\Xi \vdash \tau]\!]_\Delta$ is an Iris relation of type $(\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$ and, intuitively, it relates STLang values of type τ. The value relation is defined by induction on τ. The expression relation has the following type:

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

and it is defined independently of the value relation. In these definitions, Δ is a map from type variables to interpretation of types and closed types:

$$\Delta : \mathit{Tvar} \to (((\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}) \times \{\tau \in \mathit{Types} \mid \mathsf{FV}(\tau) = \emptyset\})$$

(the closed type is solely used to close types ρ in $\mathtt{STRef}\ \rho\ \tau$).

Intuitively, the expression relation $\mathcal{E}\,\Phi(e, e')$ holds for two expressions e and e' if e refines, or approximates, e'. That is, a reduction step taken by e can can be simulated by zero or more steps in e'. We think of e as the implementation and e' as the specification. 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':

$$(h, e) \Downarrow^\gamma_\Phi \ \triangleq\ \exists h', v.\ \langle h, e\rangle \to^*_d \langle h', v\rangle * \boxed{\bullet \mathsf{excl}(h')}^{\gamma} * \Phi(v)$$

Value relations:

$$\begin{aligned}
[\![\Xi \vdash X]\!]_\Delta &\triangleq (\Delta(X)).1\\
[\![\Xi \vdash \mathbf{1}]\!]_\Delta(v,v') &\triangleq v = v' = ()\\
[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v,v') &\triangleq v = v' \in \{\mathtt{true}, \mathtt{false}\}\\
[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v,v') &\triangleq v = v' \in \mathbb{N}\\
[\![\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\\
&\qquad [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') \wedge [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')\\
[\![\Xi \vdash \tau + \tau']\!]_\Delta(v,v') &\triangleq (\exists w, w'.\ v = \mathtt{inj}_1\, w \wedge v' = \mathtt{inj}_1\, w' \wedge [\![\Xi \vdash \tau]\!]_\Delta(w,w')) \vee\\
&\qquad (\exists w, w'.\ v = \mathtt{inj}_2\, w \wedge v' = \mathtt{inj}_2\, w' \wedge [\![\Xi \vdash \tau']\!]_\Delta(w,w'))\\
[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v,v') &\triangleq \Box\left(\forall (w,w').\ [\![\Xi \vdash \tau]\!]_\Delta(w,w') \Rightarrow \mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta(v\ w, v'\ w')\right)\\
[\![\Xi \vdash \forall X.\,\tau]\!]_\Delta(v,v') &\triangleq \Box\Big(\forall f, \tau'.\ \mathsf{FV}(\tau) = \emptyset \wedge \mathsf{persistent}(f) \Rightarrow\\
&\qquad \mathcal{E}\,[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f,\tau')}(v\ _, v'\ _)\Big)\\
[\![\Xi \vdash \mu X.\,\tau]\!]_\Delta(v,v') &\triangleq \mu f.\ \exists w, w'.\ v = \mathtt{fold}\, w \wedge v' = \mathtt{fold}\, w' \wedge\\
&\qquad \triangleright [\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto (f, \mathrm{close}(\Delta, \mu X.\,\tau))}(w,w')\\
[\![\Xi \vdash \mathsf{STRef}\ \rho\ \tau]\!]_\Delta(v,v') &\triangleq \exists \ell, \ell', \gamma_{bij}, \gamma_{rel}, \gamma_{pred}.\ v = \ell \wedge v' = \ell' \wedge \boxed{\circ\, \mathrm{close}(\Delta,\rho) \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} *\\
&\qquad \boxed{\circ\, \mathrm{ag}(\ell,\ell')}^{\gamma_{bij}} * \boxed{\circ\, (\ell,\ell') \mapsto \mathrm{ag}(\gamma_{pred})}^{\gamma_{rel}} * \gamma_{pred} \Longmapsto [\![\Xi \vdash \tau]\!]_\Delta\\
[\![\Xi \vdash \mathsf{ST}\ \rho\ \tau]\!]_\Delta(v,v') &\triangleq \forall \gamma_h, \gamma_h', h_1'.\\
&\qquad \left\{\boxed{\bullet\, \mathrm{excl}(h_1')}^{\gamma_h'} * \mathrm{regions} * \mathrm{region}(\mathrm{close}(\Delta,\rho), \gamma_h, \gamma_h')\right\}\\
&\qquad \mathtt{runST}\ \{v\}\\
&\qquad \left\{w.\ (h_1', \mathtt{runST}\ \{v'\}) \Downarrow^{\gamma_h'}_{[\![\Xi \vdash \tau]\!]_\Delta(w,\cdot)} * \mathrm{region}(\mathrm{close}(\Delta,\rho), \gamma_h, \gamma_h')\right\}_{\gamma_h}
\end{aligned}$$

Expression relation:

$$\mathcal{E}\,\Phi(e,e') \triangleq \forall \gamma_h, \gamma_h', h_1'.\ \left\{\boxed{\bullet\, \mathrm{excl}(h_1')}^{\gamma_h'} * \mathrm{regions}\right\} e \left\{w.\ (h_1', e') \Downarrow^{\gamma_h'}_{\Phi(w,\cdot)}\right\}_{\gamma_h}$$

Environment relation:

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

Logical relatedness:

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

Closing types:

$$\mathrm{close}(\Delta, \rho) \triangleq \rho[(\Delta\vec{X}).2/\vec{X}]$$

closes the (open) type ρ using the closed types stored in Δ

Fig. 9. Binary logical relation

This assertion says that there exists a *deterministic* reduction from (h, e) to (h', v), the resulting heap h' is owned and the value satisfies Φ.

The deterministic reduction relations, $\rightarrow_d$ and $\rightsquigarrow_d$, are defined by the same inference rules as $\rightarrow$ and $\rightsquigarrow$, except that the only non-deterministic rule, Alloc, is replaced by a deterministic one:

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

The requirement that the reduction on the specification side is *deterministic* is used crucially in the proofs of the purity properties in §5. We emphasize that even with this requirement, we can still prove that logical relatedness implies contextual refinement (without requiring that STLang uses deterministic reductions).

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, $\boxed{\bullet\,\mathrm{excl}(h_1')}^{\gamma_h'}$, and an assertion regions, which keeps track of all allocated regions (explained later), and 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 implementation and specification side reductions are universally quantified. Before discussing how regions are maintained, we describe how we relate values.

For ground types, $[\![\Xi \vdash \tau]\!]_\Delta$ is simply the identity relation. We say that pairs are related if they are related pointwise. Functions are related if they map related arguments to related results. As usual, the value interpretation for the polymorphic type $\forall X.\,\tau$ is defined by universal quantification over any predicate (and a closed type in our case). The value interpretation of the recursive type $\mu X.\,\tau$ is simply defined as the (guarded by $\triangleright$) fixpoint of the value interpretation of τ.

To complete the description of the expression relation and to explain the value interpretation of the types $\mathtt{STRef}\ \rho\ \tau$ and $\mathtt{ST}\ \rho\ \tau$, we now first describe the region abstraction intuitively and then we show how it is defined precisely.

Regions, intuitively. Conceptually, we take a region $\mathcal{R}$ to be a collection of pairs of locations, in one-to-one correspondence, together with an assignment of a semantic predicate $\mathcal{R}_{(\ell,\ell')}$ to each pair of locations (ℓ, ℓ') in the region. The idea then is that an implementation-side heap h and a specification-side heap h' satisfies a region $\mathcal{R}$, written $(h, h') \models \mathcal{R}$ if, for any pair of locations (ℓ, ℓ') in $\mathcal{R}$ we have values v and v', such that $h(\ell) = v$ and $h'(\ell') = v'$ and such that the values v and v' are related by the relation ascribed by the region $\mathcal{R}$, i.e., $\mathcal{R}_{(\ell,\ell')}(v, v)$.

Notice that the one-to-one correspondence is necessary because we want to compare 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.

One of the key ideas of our logical relation is that we tie each type ρ in a monad type (ρ in $\mathtt{ST}\ \rho\ \tau$) to one of these semantic regions $\mathcal{R}$. Let us write $\mathrm{region}(\mathcal{R}, \rho)$ to say that $\mathcal{R}$ is the semantic region tied to ρ. With this in mind, the intuitive definitions of the value relations for $\mathtt{STRef}\ \rho\ \tau$ and $\mathtt{ST}\ \rho\ \tau$ should then be as follows:

- intuitive definition of $[\![\Xi \vdash \mathtt{STRef}\ \rho\ \tau]\!]_\Delta(\ell, \ell')$: there is a semantic region $\mathcal{R}$ such that $\mathrm{region}(\mathcal{R}, \rho)$, $(\ell, \ell') \in \mathrm{dom}(\mathcal{R})$ and $\mathcal{R}_{(\ell,\ell')} = [\![\Xi \vdash \tau]\!]_\Delta$
- intuitive definition of $[\![\Xi \vdash \mathtt{ST}\ \rho\ \tau]\!]_\Delta(v, v')$: For any h_1, h_2, v and h_1' such that $(h_1, h_1') \models \mathcal{R}$ and $\langle h_1, \mathtt{runST}\ \{v\}\rangle \rightarrow^* \langle h_2, w\rangle$ then there is a heap h_2' and a value w' and a semantic region $\mathcal{R}' \supseteq \mathcal{R}$ such that *afterwards* we have $\langle h_1', \mathtt{runST}\ \{v'\}\rangle \rightarrow_d^* \langle h_2', w'\rangle$ and $\mathrm{region}(\mathcal{R}', \rho)$

In the intuitive definition of $\mathtt{ST}\ \rho\ \tau$, the word *afterwards* is supposed to refer an application of the future modality. Notice that the semantic region $\mathcal{R}$ is updated to $\mathcal{R}' \supseteq \mathcal{R}$ after v and v' have been computed. This is, of course, because computations can allocate memory and hence new regions can be allocated. However, the computations

cannot change the semantics of already existing locations. The relatedness of locations at a reference type $\mathtt{STRef}\ \rho\ \tau$ requires that the locations are related in the semantic region associated (at the time) to the type ρ.

In a nutshell, the semantic region tied to a type ρ is "dynamic" while the semantics of the reference type $\mathtt{ST}\ \rho\ \tau$ says that the related locations are part of the semantic region tied to ρ with the appropriate interpretation. These are exactly the kinds of properties that Iris predicates involving ownership are designed to capture.

Regions in Iris, concretely. In order to concretely represent semantic regions, 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. Formally, we use the following two monoids:

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

To tie the two monoids (the semantic region) to a closed type ρ we use a third monoid:

$$\mathsf{Reg} \triangleq \textsc{Auth}(\{\tau \in \mathit{Types} \mid \mathsf{FV}(\tau) = \emptyset\} \xrightarrow{\text{fin}} (\textsc{Ag}(\mathit{Names} \times \mathit{Names})))$$

We use a ghost variable γ_{reg} for this last monoid. In particular, the ownership $\boxed{\circ\, \rho \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ indicates that the semantic region tied to ρ is represented by two ghost variables named γ_{bij} and γ_{rel}, for Bij and Rel respectively. Notice that this ownership of $\boxed{\circ\, \rho \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$ is duplicable and also, due to the properties of the agreement monoid, we have that the semantic region tied to ρ is uniquely defined. More formally,

$$\boxed{\circ\, \rho \mapsto \mathsf{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}} * \boxed{\circ\, \rho \mapsto \mathsf{ag}(\gamma'_{bij}, \gamma'_{rel})}^{\gamma_{reg}} \vdash \gamma_{bij} = \gamma'_{bij} \wedge \gamma_{rel} = \gamma'_{rel} \tag{2}$$

The region predicate below keeps track of all the allocated regions and maintains the fact that the bijection part of each region is indeed a bijection and that every pair in this bijection is indeed part of the semantic regions relation.

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

Notice that due to the use of the invariant the regions predicate is persistent. As discussed earlier, we use ghost state to keep track of the implementation and specification side heaps. The following predicate $\mathsf{region}(\rho, \gamma_h, \gamma'_h)$ intuitively says that the two heaps are kept track of by using a piece of ghost state named γ_h (the left hand side heap) and γ'_h (the right hand side heap) and that they satisfy the semantic region tied to ρ.

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

For proving the compatibility lemmas for allocation of new reference cells we need to show that the monoids representing the regions can be appropriately updated. Proving the compatibility lemmas for reading from and writing to references, on the other hand, requires one to obtain $\boxed{\circ\, \ell \mapsto \mathsf{ex}(v)}^{\gamma_h}$ and $\boxed{\circ\, \ell' \mapsto \mathsf{ex}(v')}^{\gamma'_h}$ for some value v and v'. These ghost-resources can be recovered from the interpretation of references $[\![\Xi \vdash \mathtt{STRef}\ \rho\ \tau]\!]_\Delta(\ell, \ell')$, the regions predicate and $\mathsf{region}(\rho, \gamma_h, \gamma'_h)$ by using the rules for ownership and invariants stated in §3. Here we only briefly sketch the proof of the compatibility lemma for `runST`. Detailed proofs for all the compatibility lemmas can be found in the accompanying (Technical Report 2017).

Lemma 4.1 (Compatibility for runST). *Suppose* $\Xi, X \mid \Gamma \vDash e \preceq_{\log} e' : \mathsf{ST}\ X\ \tau$ *and* $\Xi \vdash \tau$. *Then*

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

Proof. We prove a slightly simpler but sufficiently illuminating version for the sake of conciseness. We show that if $\forall f, \tau'.\ [\![\Xi, X \vdash \mathsf{ST}\ X\ \tau]\!]_{\Delta, X \mapsto (f,\tau')}(v, v')$ then $\mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(\mathtt{runST}\ \{v\}, \mathtt{runST}\ \{v'\})$. Let us assume that we have γ_h, γ_h', regions, h_1' and $\boxed{\bullet \mathrm{excl}(h_1')}^{\gamma_h'}$. We are to show:

$$\mathsf{IC}^{\gamma_h}\ \mathtt{runST}\ \{v\}\ \left\{w.\ \exists h_2', w'.\ \left\langle h_1', \mathtt{runST}\ \{v'\}\right\rangle \rightarrow_d^* \left\langle h_2', w'\right\rangle * \boxed{\bullet \mathrm{excl}(h_2')}^{\gamma_h'} * [\![\Xi \vdash \tau]\!]_\Delta(w, w') * \mathrm{regions}\right\}$$

Now we can allocate $\boxed{\bullet \emptyset}^{\gamma_{bij}}$, $\boxed{\circ \emptyset}^{\gamma_{bij}}$, $\boxed{\bullet \emptyset}^{\gamma_{rel}}$ and $\boxed{\circ \emptyset}^{\gamma_{rel}}$ for fresh γ_{bij} and γ_{rel}. By having the full part in the regions predicate (under the invariant), we can allocate $\boxed{\circ\ \rho \mapsto \mathrm{ag}(\gamma_{bij}, \gamma_{rel})}^{\gamma_{reg}}$, for a fresh closed type ρ, and update the regions predicate appropriately (by opening the invariant). This allows us to get $\mathrm{region}(\rho, \gamma_h, \gamma_h')$. Notice that the big separating conjunction in $\mathrm{region}(\rho, \gamma_h, \gamma_h')$ would be a big separating conjunction over the empty set and thus is trivial. Now we can instantiate our assumption to get $[\![\Xi, X \vdash \mathsf{ST}\ X\ \tau]\!]_{\Delta, X \mapsto ([\![\cdot + 1]\!], \rho)}(v, v')$.

Now using Lemma 3.3 Case 3 we get that there are h_2', w' such that we have $\boxed{\bullet \mathrm{excl}(h_2')}^{\gamma_h'}$, $\left\langle h_1', \mathtt{runST}\ \{v'\}\right\rangle \rightarrow_d^* \left\langle h_2', w'\right\rangle$ and $[\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto ([\![\cdot + 1]\!], \rho)}(w, w')$ and subsequently we have to show

$$\Rrightarrow \exists h_2', w'.\ \left\langle h_1', \mathtt{runST}\ \{v'\}\right\rangle \rightarrow_d^* \left\langle h_2', w'\right\rangle * \boxed{\bullet \mathrm{excl}(h_2')}^{\gamma_h'} * [\![\Xi \vdash \tau]\!]_\Delta(w, w') * \mathrm{regions}$$

The only thing that we don't immediately have is $[\![\Xi \vdash \tau]\!]_\Delta(w, w')$. However it follows from $\Xi \vdash \tau$ that $[\![\Xi \vdash \tau]\!]_\Delta(w, w') \dashv\vdash [\![\Xi, X \vdash \tau]\!]_{\Delta, X \mapsto ([\![\cdot + 1]\!], \rho)}(w, w')$. □

Notice that in this proof we start out with two totally unrelated heaps for the specification and the implementation side (universally quantified inside the IC predicate). We then establish a *trivial* relation between them by creating a new *empty* region and maintain that 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.

Theorem 4.2 (Fundamental theorem). $\Xi \mid \Gamma \vdash e : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\log} e : \tau$

Proof. Follows from compatibility lemmas by induction on typing derivation $\Xi \mid \Gamma \vdash e : \tau$. □

Theorem 4.3 (Adequacy of logical relation). $\mathcal{E}\,[\![\Xi \vdash \tau]\!]_\Delta\,(e, e') \wedge \forall h, h'.\ (h, e) \downarrow \Rightarrow (h', e') \downarrow$

Proof. Follows from the definition of expression relation and the soundness of the Iris logic itself. □

Theorem 4.4 (Congruence of logical relation). $\Xi \mid \Gamma \vDash e \preceq_{\log} e' : \tau \Rightarrow \forall C.\ C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau') \Rightarrow \Xi' \mid \Gamma' \vDash C[e] \preceq_{\log} C[e'] : \tau'$

Proof. Follows from compatibility lemmas and the fundamental theorem by induction on derivation of $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')$. □

Theorem 4.5 (Soundness of logical relation). $\Xi \mid \Gamma \vDash e \preceq_{\log} e' : \tau \Rightarrow \Xi \mid \Gamma \vDash e \preceq_{\mathrm{ctx}} e' : \tau$

5 PROVING CONTEXTUAL REFINEMENTS AND EQUIVALENCES

In this section we give proof sketches of the contextual equivalences and refinements shown in Figure 2. Detailed and formal proofs of these theorems, and the results in Figure 3, can be found in the (Technical Report 2017).

Lemma 5.1 (Neutrality). $\Xi \mid \Gamma \vDash e \preceq_{\mathrm{ctx}} () : \mathbf{1}$

Proof Sketch. By the fundamental theorem of logical relations, we get that $\Xi \mid \Gamma \vDash e \preceq_{\log} e : 1$. Therefore, if $\langle h_1, e\rangle$ reduces to a value then it is related in the value relation for the unit type, and since () is already a value, the result follows. □

Lemma 5.2 (Commutativity). $\Xi \mid \Gamma \vDash \texttt{let}\, x = e_2\, \texttt{in}\, (e_1, x) \;\approx_{\text{ctx}}\; (e_1, e_2) \;:\; \tau_1 \times \tau_2$

Proof Sketch. The proof of both sides of the contextual equivalence are similar. We only discuss the left-to-right refinement by showing the following logical relatedness, under the assumption that $e_1 \preceq_{\log} e_1'$ and $e_2 \preceq_{\log} e_2'$:

$$\texttt{let}\, x = e_2\, \texttt{in}\, (e_1, x) \;\preceq_{\log}\; (e_1', e_2') \;:\; \tau_1 \times \tau_2$$

Assume that we own an implementation heap $\boxed{\bullet \text{excl}(h_1)}^{\gamma_h}$. We know that for h_1 the expression $\texttt{let}\, x = e_2\, \texttt{in}\, (e_1, x)$ reduces to a value therefore there must be a h_2, h_3, v_1, v_2 such that $\langle h_1, e_2\rangle \to^* \langle h_2, v_2\rangle$ and $\langle h_2, e_1\rangle \to^* \langle h_3, v_1\rangle$. We have to show three things: we can obtain $\boxed{\bullet \text{excl}(h_3)}^{\gamma_h}$, $\langle h_1', (e_1', e_2')\rangle$ deterministically reduces to a value and the two values are related. Defining contexts and updating the specification heap is straightforward and elided from this sketch.

The trick is now to allocate a new ghost-variable γ_1 as so: $\boxed{\bullet \text{excl}(h_2)}^{\gamma_1}$. Using this and $e_1 \preceq_{\log} e_1'$ with $\langle h_2, e_1\rangle \to^* \langle h_3, v_1\rangle$ we obtain $\langle h_1', e_1'\rangle \to_d^* \langle h_2', v_1'\rangle$, $[\![\Xi \vdash \tau_1]\!]_\Delta(v_1, v_1')$ and $\boxed{\bullet \text{excl}(h_3)}^{\gamma_1}$. Furthermore, by some resource reasoning, we know $h_3 = h_2 \uplus h_A$ for some h_A (intuitively, this is because e_1 cannot modify h_2, so it can only extend it).

Now, with the original heap $\boxed{\bullet \text{excl}(h_1)}^{\gamma_h}$, and by $e_2 \preceq_{\log} e_2'$ with $\langle h_1, e_2\rangle \to^* \langle h_2, v_2\rangle$ we obtain $\langle h_2', e_2'\rangle \to_d^* \langle h_3', v_3'\rangle$, $[\![\Xi \vdash \tau_2]\!]_\Delta(v_2, v_2')$ and $\boxed{\bullet \text{excl}(h_2)}^{\gamma_h}$. By composing the two reductions, we obtain a deterministic reduction on the specification side to a value (v_1', v_2'), which is related to (v_1, v_2) on the implementation side. Finally, it follows directly from $h_3 = h_2 \uplus h_A$ that we can obtain $\boxed{\bullet \text{excl}(h_3)}^{\gamma_h}$ from $\boxed{\bullet \text{excl}(h_2)}^{\gamma_h}$.

□

Lemma 5.3 (Idempotency). $\Xi \mid \Gamma \vDash \texttt{let}\, x = e\, \texttt{in}\, (x, x) \;\approx_{\text{ctx}}\; (e, e) \;:\; \tau \times \tau$

Proof Sketch. The proof is similar to that presented in Lemma 3.4. For all the details we refer to the (Technical Report 2017). □

Lemma 5.4 (Rec Hoisting).

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

Proof Sketch. Proving rec hoisting for a language with state is no mean feat, which is also true for STLang. This is particularly difficult in step-indexed models, such as ours, since the steps do not match up for expressions under the function. For a lengthier discussion about these difficulties, we refer to the (Technical Report 2017).

In this paper we employ a neat trick to accomplish proving rec hoisting. We, in fact, prove three different contextual refinements, such that their composition gives us the desired contextual refinement above. These contextual refinements are as follows:

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

We prove case (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 right hand side. Notice, that we cannot assert relatedness to the original computed value on the right, but this is not necessary as the value on the left is simply *thrown away*. We prove case (c) also by the corresponding logical relatedness which is rather trivial to prove.

To prove case (b) we show that left hand side and right hand side are logically related under a slightly stronger logical relation; $\preceq_{\log}^{NN}$. The NN logical relation is defined entirely similarly to the primary logical relation except that the right hand side is required to deterministically reduce to a value in the *same number of steps* as the left hand side. Notice that the proofs of the NN logical relation being congruent (compatibility lemmas) and adequate, they are similar to those of the primary logical relation, there is simply a bit of care needed to assure that the number of steps on the right hand side is the same as the ones on the left.

Formally, for case (b) we have to prove

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

The crucial step in this proof is to prove that all instances of e_1 reduce in the same number of steps on the left and right. However, this is not too difficult to see.

At the beginning of the proof we have that e_1 outside the function, reduces in n steps on the left under a heap h_1, and therefore also deterministically reduces on the right in n steps under the heap h_1'. Similarly, inside the function, we know that e_1 reduces in m steps under a heap h_3 and therefore so does deterministically the e_1 on the right hand side inside the function under a heap h_3'.

Analogously to the trick for proving commutativity, we can allocate two new ghost-variables γ_1 and γ_2 initialized to the left hand side heap h_3 and the right hand side heap h_1'. This gives us that e_1 outside the function on the right under h_1' also reduces deterministically to a value in m steps. By the reduction on the right hand side being deterministic, the two executions of e_1 starting from h_1' should take the same number of steps. From this we know that $m = n$, which allows us to conclude that the newly computed value of e_1 on the right hand side inside the function is related to the computation of e_1 on the left hand side outside the function. This concludes the proof. □

Lemma 5.5 (Λ Hoisting). $\Xi \mid \Gamma \vDash \texttt{let}\, y = e_1\, \texttt{in}\, \Lambda\, e_2 \;\preceq_{\text{ctx}}\; \Lambda\, (\texttt{let}\, y = e_1\, \texttt{in}\, e_2) \;:\; \forall X.\, \tau$

Proof Sketch. The proof of this lemma is very similar to the proof of rec hoisting above. □

Lemma 5.6 (η expansion for rec). $\Xi \mid \Gamma \vDash e \;\preceq_{\text{ctx}}\; \texttt{rec}\, f(x) = (e\; x) \;:\; \tau_1 \to \tau_2$

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

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

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

Lemma 5.7 (η expansion for Λ). $\Xi \mid \Gamma \vDash e \;\preceq_{\text{ctx}}\; \Lambda\, (e\; \_) \;:\; \forall X.\, \tau$

Proof Sketch. The proof of this lemma is very similar to the proof of η expansion for rec above. We simply use Λ hoisting instead of rec hoisting. □

Lemma 5.8 (β reduction for λ).

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

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

$$\text{Case}\; \frac{\Xi \mid \Gamma, x : \tau \vdash e : \tau_i \qquad i \in \{1, 2\}}{\Xi \mid \Gamma, x : \tau \vdash \texttt{inj}_i\, e : \tau_1 + \tau_2}:$$

IH: $\Xi \mid \Gamma \vDash (\lambda\, x.\, e)\ e_2 \preceq_{\text{ctx}} e[e_2/x] : \tau_i$

We have to show that $\Xi \mid \Gamma \vDash (\lambda\, x.\, \texttt{inj}_i\ e)\ e_2 \preceq_{\text{ctx}} (\texttt{inj}_i\ e)[e_2/x] : \tau_1 + \tau_2$.

Notice that it is easy to prove (using the fundamental theorem) that

$$\Xi \mid \Gamma \vDash (\lambda\, x.\, \texttt{inj}_i\ e)\ e_2 \preceq_{\text{log}} \texttt{inj}_i\ ((\lambda\, x.\, e)\ e_2) : \tau_1 + \tau_2$$

The final result follows by the induction hypothesis, transitivity of contextual refinement and the fact that contextual refinement is a congruence relation.

Case $\dfrac{\Xi \mid \Gamma, x : \tau, y : \tau_1, f : \tau_1 \to \tau_2 \vdash e : \tau_2}{\Xi \mid \Gamma, x : \tau \vdash \texttt{rec}\, f(y) = e : \tau_1 \to \tau_2}$:

IH: $\Xi \mid \Gamma, y : \tau_1, f : \tau_1 \to \tau_2 \vDash (\lambda\, x.\, e)\ e_2 \preceq_{\text{ctx}} e[e_2/x] : \tau_2$

We have to show that $\Xi \mid \Gamma \vDash (\lambda\, x.\, (\texttt{rec}\, f(y) = e))\ e_2 \preceq_{\text{ctx}} (\texttt{rec}\, f(y) = e)[e_2/x] : \tau_1 \to \tau_2$ or equivalently (by just massaging the terms) $\Xi \mid \Gamma \vDash \texttt{let}\, x = e_2\, \texttt{in}\, (\texttt{rec}\, f(y) = e) \preceq_{\text{ctx}} (\texttt{rec}\, f(y) = e[e_2/x]) : \tau_1 \to \tau_2$.

Notice that the following is instance of rec-hoisting.

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

This latter contextual refinement is equivalent to (by massaging the terms) to the following.

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

The final result follows by the induction hypothesis, transitivity of contextual refinement and the fact that contextual refinement is a congruence relation.

□

Lemma 5.9 (β reduction for Λ). $\Xi \mid \Gamma \vDash (\Lambda\, e)\ \_ \preceq_{\text{ctx}} e : \tau'$

Proof Sketch. We prove the corresponding logical relatedness which is rather trivial. □

Lemma 5.10 (Rec unfolding).

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

Proof Sketch. We prove the corresponding logical relatedness which is easy. □

Lemma 5.11 (β reduction for rec).

$$\Xi \mid \Gamma \vDash (\texttt{rec}\, f(x) = e_1)\ e_2 \ \preceq_{\text{ctx}} e_1[e_2, (\texttt{rec}\, f(x) = e_1)/x, f] \ : \ \tau$$

Proof Sketch. Follows by rec unfolding, β reduction for λ and congruence. □

Lemma 5.12 (Contextual Equivalences for stateful computations).

Proof. All of the contextual equivalences in Figure 3 follow from their corresponding logical relatednesses (separately for each direction). □

6 RELATED WORK

The most closely related work is the original seminal work of Launchbury and Peyton Jones (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 (Moggi and Sabry 2001) showed type soundness of a calculi with `runST`-like constructs, both for a call-by-value language (as we consider here) and for a lazy language. In contrast to our work, they focused on type soundness and did not prove contextual equivalences justifying that the pure language remains pure. They 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).*"

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 (Gifford and Lucassen 1986). This connection was formalized by Semmelroth and Sabry (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 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 (Thamsborg and Birkedal 2011) to a language with higher-order store and 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 work 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, and we use that in the proofs of contextual refinements in §5. (There are other more technical differences, which we discuss in the Technical Report (2017).)

7 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, our model 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.

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.

Other future work includes giving a full Coq formalization of the model construction and all our proofs. So far, we have formalized the basic building blocks of the present work, e.g., the future modality and its properties, on top of the Iris formalization in the Coq proof assistant (Krebbers et al. 2017b).

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, Ale 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.

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

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.

Technical Report. 2017. *A Logical Relation for Monadic Encapsulation of State: Proving contextual equivalences in the presence of* `runST`. Technical Report. Technical Appendix.

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