A Concurrent Logical Relation

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg

IT University of Copenhagen, Denmark
`{birkedal, fisi, thamsborg}@itu.dk`

Abstract

We present a logical relation for showing the correctness of program transformations based on a new type-and-effect system for a concurrent extension of an ML-like language with higher-order functions, higher-order store and dynamic memory allocation.

We show how to use our model to verify a number of interesting program transformations that rely on effect annotations. In particular, we prove a Parallelization Theorem, which expresses when it is sound to run two expressions in parallel instead of sequentially. The conditions are expressed solely in terms of the types and effects of the expressions. To the best of our knowledge, this is the first such result for a concurrent higher-order language with higher-order store and dynamic memory allocation.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases verification, logical relation, concurrency, type and effect system

Digital Object Identifier 10.4230/LIPIcs.CSL.2012.107

1 Introduction

Relational reasoning about program equivalence is useful for reasoning about the correctness of program transformations, data abstraction (representation independence), compiler correctness, etc. The standard notion of program equivalence is contextual equivalence and in recent years, there have been many improvements in reasoning methods for higher-order ML-like languages with general references, based on bisimulations, e.g., [17, 23, 25], traces [18], game semantics [21], and Kripke logical relations, e.g., [1, 2, 8, 12].

In this paper we present the first Kripke logical relation for reasoning about equivalence of a *concurrent* higher-order ML-like language with higher-order store and dynamic memory allocation.

To state and prove useful equivalences about concurrent programs, it is necessary to have some way of restricting the contexts under which one proves equivalences. This point was made convincingly in the recent paper by Liang et. al. [19], who presented a rely-guarantee-based simulation for verifying concurrent program transformations for a first-order imperative language (with first-order store). Here is a very simple example illustrating the point. Consider two expressions

$$e_1 \equiv x := 1; y := 1 \quad \text{and} \quad e_2 \equiv y := 1; x := 1.$$

Here x and y are variables of type `refint`. The expressions e_1 and e_2 are not contextually equivalent. (To see why, consider expression $e_3 \equiv x := 0;\ y := 0$, and note that running e_1 in parallel with e_3 may result in a state with $!x = 0$ and $!y = 1$, but that cannot be the case when we run e_2 in parallel with e_3.) The issue is, of course, that the context may also modify the references x and y. On the other hand, if we know that no other threads have access to x or y, then it should be the case that e_1 and e_2 are equivalent. We can express this restriction on the contexts using a refined region-based type-and-effect system.

We first recall that a type-and-effect system is a type system that classifies programs according to which side effects the programs may have. A variety of effect systems have been proposed for higher-order programming languages, e.g., [15, 20, 27], see [16] for a recent overview. Effect systems can often be understood as specifying the results of a static analysis, in the sense that it is possible to automatically infer types and effects. Effect systems can be used for different purposes: they were originally proposed by Lucassen and Gifford [20] for parallelization purposes but they have also, e.g., been used as the basis for implementing ML using a stack of regions for memory management [27, 9]. In a recent series of papers, Benton et. al. have argued that another important point of effect systems is that they can be used as the basis for effect-based program transformations, e.g., compiler optimizations, [6, 5, 3, 4], see also [26]. The idea is that certain program transformations are only sound under additional assumptions about which effects program phrases may, or rather may not, have.

Now, returning to our example, we refine the types of x and y to be $\mathsf{ref}_\rho\mathsf{int}$ and $\mathsf{ref}_\sigma\mathsf{int}$, respectively. Intuitively, this expresses that x and y are references in different regions, but it does not put any restrictions on whether other threads may access x or y. Thus, when we type e_1 and e_2 we will use two contexts of region variables, one for public regions that can be used by other concurrently running threads, and one for private regions that are under the control of the present thread. This idea is inspired by recent work on concurrent separation logic, e.g., [22, 11, 29, 13]. We use a vertical bar to separate public and private regions: the typing context

$$\rho, \sigma \mid \emptyset \mid x : \mathsf{ref}_\rho\mathsf{int}, y : \mathsf{ref}_\sigma\mathsf{int}$$

expresses that ρ and σ are public regions, whereas the typing context

$$\emptyset \mid \rho, \sigma \mid x : \mathsf{ref}_\rho\mathsf{int}, y : \mathsf{ref}_\sigma\mathsf{int}$$

expresses that ρ and σ are private regions. The expressions e_1 and e_2 are well-typed in the latter context and, with this refined typing, they are indeed contextually equivalent, because our type-and-effect system guarantees that no well-typed context can access regions ρ or σ. (The expressions are also well-typed in the former context, but not contextually equivalent with that refined typing.)

In this paper we present a step-indexed Kripke logical relations model of a type-and-effect system with public and private regions for a concurrent higher-order language with general references. Our model is constructed over the operational semantics of the programming language, and builds on recent work by Thamsborg and Birkedal on logical relations for the sequential sub-language [26]. Note that the type-and-effect annotations are just annotations; the operational semantics of the language is standard and regions only exist in our semantic model, not in the operational semantics.

As an important application of our model we prove a Parallelization Theorem, which expresses when it is sound to run two expressions in parallel instead of sequentially. To the best of our knowledge, this is the first such result for a higher-order language with higher-order store and dynamic memory allocation. Here is a very simple instance of the theorem. Consider two expressions

$$e_1 \equiv y := !x + !y \quad \text{and} \quad e_2 \equiv z := !x + !z,$$

each well-typed in a context

$$\emptyset \mid \rho_x, \rho_y, \rho_z \mid x : \mathsf{ref}_{\rho_x}\mathsf{int}, y : \mathsf{ref}_{\rho_y}\mathsf{int}, z : \mathsf{ref}_{\rho_z}\mathsf{int},$$

i.e., where x, y, and z are references in distinct private regions. In this context, running e_1 and e_2 sequentially is contextually equivalent to running e_1 and e_2 in parallel. Intuitively, this also makes sense: e_1 and e_2 update references in distinct regions, and it is unproblematic that they both read (but not write) from the same region.

As mentioned, this was a simple instance of the Parallelization Theorem. We stress that the theorem is expressed solely in terms of the type and efffects of the expressions e_1 and e_2, so a compiler may automatically infer that it is safe to parallelize two expressions by looking at the inferred effect types, and without reasoning about all interleavings. Moreover, the theorem applies to contexts and expressions with general higher types (not just with references to integers and unit types). Note that the distinction between private and public regions is also crucial here (parallelization would not be sound if the effects of the expressions were on public regions).

Our type-and-effect system crucially also includes a region-masking rule. Traditionally, this rule has been used to hide local effects on regions, which makes it possible to view a computation as pure even if it uses effects locally and makes the effect system stronger, in the sense that it can justify more program transformations. Here we also observe that the masking rule can be used for introducing private regions, since the masking rule intuitively guarantees that effects on a region are not leaked to the context. It is well-known that region-masking makes the model construction for a sequential language technically challenging, see the extensive discussion in [26]. Here it is yet more challenging because of concurrency; we explain how our model ensures soundness of the masking rule in Section 3.

The extension with concurrency also means that when we define the logical relation for contextual approximation and relate two computations e_1 and e_2, we cannot simply require relatedness after e_1 has completed evaluation (as in the sequential case), since other threads should be allowed to execute as well. We explain our approach to relating concurrent computations in Section 3; it is informed by recent soundness proofs of unary models of concurrent separation logic [30, 10].

Another challenge arises from the fact that since our language includes dynamically allocated general references, the existence of the logical relation is non-trivial; in particular, the set of Kripke worlds must be recursively defined. Here we build on our earlier work [7] and define the worlds as a solution to a recursive metric-space equation. Indeed, to focus on the essential new aspects due to the extension with concurrency, we deliberately choose to use the exact same notion of worlds as we used for the sequential sub-language in [26]. In the same vein, we here consider a monomorphically typed higher-order programming language with general references, but leave out universal and existential types as well as recursive types. However, we want to stress that since our semantic techniques (step-indexed Kripke logical relations over recursively defined worlds) do indeed scale well to universal, existential, and recursive types, e.g. [7, 12], it is possible to extend our model to a language with such types. We conjecture that it is also possible to extend our model to richer effect systems involving region and effect polymorphism, but we have not done so yet.

All proofs are deferred to the long version of the paper; it can be found online at the following address: `www.itu.dk/people/birkedal/papers/longsamba.pdf`.

2 Language and Typing

We consider a standard call-by-value lambda calculus with general references, and extended with parallel composition and an atomic construct. We assume countably infinite, pairwise disjoint sets of *region variables* $\mathcal{RV}$ (ranged over by ρ), *locations* $\mathcal{L}$ (ranged over by l) and

$$
\begin{array}{rcl}
\pi & ::= & rd_\rho \mid wr_\rho \mid al_\rho \\
\varepsilon & ::= & \pi_1, \dots, \pi_n \\
\tau & ::= & 1 \mid \mathsf{int} \mid \tau_1 \times \tau_2 \mid \mathsf{ref}_\rho\, \tau \\
 & & \mid \tau_1 \to^{\Pi,\Lambda}_{\varepsilon} \tau_2 \\
v & ::= & x \mid \langle\rangle \mid \langle v_1, v_2 \rangle \\
 & & \mid \mathsf{fun}\, f(x).e \mid l \\
e & ::= & v \mid \mathsf{proj}_i\, v \mid v\, e \mid \mathsf{ref}\, v \mid !\, v \\
 & & \mid v_1 := v_2 \mid \mathsf{par}\ e_1\ \mathsf{and}\ e_2 \\
 & & \mid \mathsf{cas}\ (v_1, v_2, v_3) \mid \mathsf{atomic}\ e \\
E & ::= & [] \mid v\, E \mid \mathsf{par}\ E\ \mathsf{and}\ e_2 \\
 & & \mid \mathsf{par}\ e_1\ \mathsf{and}\ E
\end{array}
$$

Figure 1 Syntax.

$$
\begin{array}{rcl}
(E[\mathsf{proj}_i\, \langle v_1, v_2\rangle] \mid h) & \longmapsto & (E[v_i] \mid h) \\
(E[(\mathsf{fun}\, f(x).e)\, v] \mid h) & \longmapsto & (E[e[\mathsf{fun}\, f(x).e/f, v/x]] \mid h) \\
(E[\mathsf{ref}\, v] \mid h) & \longmapsto & (E[l] \mid h[l \mapsto v]) \text{ if } l \notin dom(h) \\
(E[l := v] \mid h) & \longmapsto & (E[\langle\rangle] \mid h[l{:=}v]) \text{ if } l \in dom(h) \\
(E[!\, l] \mid h) & \longmapsto & (E[h(l)] \mid h) \quad \text{if } l \in dom(h) \\
(E[\mathsf{par}\ v_1\ \mathsf{and}\ v_2] \mid h) & \longmapsto & (E[\langle v_1, v_2\rangle] \mid h) \\
(E[\mathsf{cas}\ (l, n_1, n_2)] \mid h) & \longmapsto & (E[1] \mid h[l{:=}n_2]) \\
 & & \text{if } l \in dom(h) \text{ and } h(l) = n_1 \\
(E[\mathsf{cas}\ (l, n_1, n_2)] \mid h) & \longmapsto & (E[0] \mid h) \\
 & & \text{if } l \in dom(h) \text{ and } h(l) \neq n_1 \\
(E[\mathsf{atomic}\ e] \mid h) & \longmapsto & (E[v] \mid h') \\
 & & \text{if } (e \mid h) \longmapsto^* (v \mid h') \\
(E[\mathsf{atomic}\ e] \mid h) & \longmapsto & (E[\mathsf{atomic}\ e] \mid h)
\end{array}
$$

Figure 2 Operational semantics.

program variables (ranged over by x, y, f). As usual, the reduction relation is between configurations, $(e \mid h) \longmapsto (e' \mid h')$ where heaps $h, h' \in \mathcal{H}$ are finite maps from locations to values. Figures 1 and 2 give the syntax and operational semantics; we denote the set of expressions $\mathcal{E}$ and the set of values $\mathcal{V}$. The evaluation contexts allow parallel evaluation inside par expressions, and there is a new primitive reduction covering the case when the two subcomputations have terminated. For technical simplicity, we allow an atomic e expression to reduce to itself, possibly introducing more divergence than the diverging behaviours of e. The syntax is kept minimal; in examples we may use additional syntactic sugar, e.g., writing let $x = e_1$ in e_2 for $(\mathsf{fun}\, f(x).e_2)\, e_1$ for some fresh f. For $e \in \mathcal{E}$, we write $\mathrm{FV}(e)$ and $\mathrm{FRV}(e)$ for the sets of free program variables and region variables, respectively; also we define $\mathrm{rds}\, \varepsilon = \{\rho \in \mathcal{RV} \mid rd_\rho \in \varepsilon\}$ and similarly for writes and allocation.

The form of the judgments of our type-and-effect system is standard with one important refinement: regions are partitioned into *public* and *private* regions, with the purpose of restricting interference from the environment. In greater detail, a typing judgement looks like this:

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

The Γ, e and τ are the usual: the *variable context* Γ assigns types to program variables in the expression e, with the resulting type of τ. To get an idea of — or rather an upper bound of — the side-effects of e, we split the heap into *regions*; these are listed in Π and Λ. We track memory accesses by adding a set ε of *effects* of the form rd_ρ, wr_ρ and al_ρ, where ρ is a region. Roughly, a computation with effect rd_ρ may read one or more locations in region ρ, and similarly for writes and allocation. This setup goes back to Lucassen and Gifford [20].

The novelty, as mentioned in the Introduction, is our partition of regions into the *public* ones Π and the *private* ones Λ. As opposed to the rest of the judgment, this public-private division does not make promises about the behavior of e. Instead, it states the expectations that e has of the environment: threads running in parallel with e may — in a well-typed manner — read, write and allocate in the public regions but must leave the private regions

$$\frac{}{\Pi \,|\, \Lambda \,|\, \Gamma, x{:}\tau \vdash x : \tau,\ \emptyset} \qquad \frac{}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \langle\rangle : \mathsf{1},\ \emptyset} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \tau_1 \times \tau_2,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{proj}_i\, v : \tau_i,\ \varepsilon}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 : \tau_1,\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_2 : \tau_2,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \langle v_1, v_2\rangle : \tau_1 \times \tau_2,\ \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma, f{:}\tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2, x{:}\tau_1 \vdash e : \tau_2,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{fun}\, f(x).e : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2,\ \emptyset}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \tau_1 \to_{\varepsilon}^{\Pi,\Lambda} \tau_2,\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau_1,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v\, e : \tau_2,\ \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \tau,\ \varepsilon \qquad \rho \in \Pi, \Lambda}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{ref}\, v : \mathsf{ref}_\rho \tau,\ \varepsilon \cup \{al_\rho\}}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 : \mathsf{ref}_\rho \tau,\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_2 : \tau,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 := v_2 : \mathsf{1},\ \varepsilon_1 \cup \varepsilon_2 \cup \{wr_\rho\}} \qquad \frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v : \mathsf{ref}_\rho \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash\, !v : \tau,\ \varepsilon \cup \{rd_\rho\}}$$

$$\frac{\Pi \,|\, \Lambda, \rho \,|\, \Gamma \vdash e : \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon - \rho}\ (\rho \notin \mathrm{FRV}(\Gamma, \tau)) \qquad \frac{\cdot \,|\, \Pi, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{atomic}\ e : \tau,\ \varepsilon}\ (\mathrm{als}\,\varepsilon \subseteq \mathrm{rds}\,\varepsilon \cap \mathrm{wrs}\,\varepsilon)$$

$$\frac{\Pi, \Lambda \,|\, \cdot \,|\, \Gamma \vdash e_1 : \tau_1,\ \varepsilon_1 \qquad \Pi, \Lambda \,|\, \cdot \,|\, \Gamma \vdash e_2 : \tau_2,\ \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{par}\ e_1\ \mathsf{and}\ e_2 : \tau_1 \times \tau_2,\ \varepsilon_1 \cup \varepsilon_2}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash v_1 : \mathsf{ref}_\rho \mathsf{int},\ \varepsilon_1 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_2 : \mathsf{int},\ \varepsilon_2 \qquad \Pi \,|\, \Lambda \,|\, \Gamma \vdash v_3 : \mathsf{int},\ \varepsilon_3}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash \mathsf{cas}\,(v_1, v_2, v_3) : \mathsf{int},\ \{wr_\rho, rd_\rho\} \cup \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}$$

$$\frac{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau_1,\ \varepsilon_1 \qquad \Pi, \Lambda \vdash \tau_1 \leq \tau_2 \qquad \varepsilon_1 \subseteq \varepsilon_2}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau_2,\ \varepsilon_2}\ (\mathrm{FRV}(\varepsilon_2) \subseteq \Pi, \Lambda)$$

$$\frac{}{\Theta \vdash \tau \leq \tau}\ (\mathrm{FRV}(\tau) \subseteq \Theta) \qquad \frac{\Theta \vdash \tau_1 \leq \tau_1' \qquad \Theta \vdash \tau_2 \leq \tau_2'}{\Theta \vdash \tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

$$\frac{\Theta \vdash \tau_1' \leq \tau_1 \qquad \Theta \vdash \tau_2 \leq \tau_2' \qquad \varepsilon_1 \subseteq \varepsilon_2 \qquad \Pi_1 \subseteq \Pi_2 \qquad \Lambda_1 \subseteq \Lambda_2}{\Theta \vdash \tau_1 \to_{\varepsilon_1}^{\Pi_1,\Lambda_1} \tau_2 \leq \tau_1' \to_{\varepsilon_2}^{\Pi_2,\Lambda_2} \tau_2'}\ (\mathrm{FRV}(\varepsilon_2), \Pi_2, \Lambda_2 \subseteq \Theta)$$

Figure 3 Typing and subtyping relations. Notice that for a typing judgement $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon$ we always have $\mathrm{FRV}(\Gamma, \tau, \varepsilon) \subseteq \Pi \cup \Gamma$.

untouched.

When running parallel threads, the private regions of the parent are shared between the children, and so are public from their point of view; this is reflected in the typing rule for parallel composition, c.f. Figure 3. Note that the parent thread only continues once both children have terminated; as a consequence, the parent regains ownership of its private regions before it goes on. Running an expression atomically temporarily makes all regions private. The side condition is a technical necessity. Finally, new, private regions are introduced by the so-called masking rule:

$$\frac{\Pi \,|\, \Lambda, \rho \,|\, \Gamma \vdash e : \tau,\ \varepsilon}{\Pi \,|\, \Lambda \,|\, \Gamma \vdash e : \tau,\ \varepsilon - \rho}\ (\rho \notin \mathrm{FRV}(\Gamma, \tau))$$

The subtraction of ρ in the conclusion removes any read, write or allocation effects tagged with ρ. The reading of the masking rule is that we make a brand new, empty region ρ for e to use, but once e has terminated we forget about ρ again; this works out since the side condition prevents e from leaking locations from ρ. Traditionally, the masking rule has been used to do memory-management [27] as well as a means of hiding local effects to facilitate effect-based program transformations [5, 26]. Here we make another use of the rule: we observe that, moreover, e cannot leak locations from ρ while running and so ρ is a *private* region for the duration of e. After all, the only means of inter-thread communication is shared memory. Note that from the perspective of the context, this rule allows to *remove* a private region, and prepare a setup for application of the parallel composition.

All the typing rules are in Figure 3. We just remark here, that reference types are tagged with the region where the location resides and that function arrows are tagged with the latent effects as well as with the public and private regions that the function expects; the latter is natural once we remember that a function is basically just a suspended, well-typed expression.

Because of the nondeterminism arising from par and shared references, the definition of contextual equivalence could take into account both may- and must-convergence. In this paper we only consider may-equivalence and formally we define (may-) contextual approximation by:

▶ **Definition 1.** $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e \lesssim_{\downarrow} e' : \tau, \varepsilon$ if and only if for all h and C typed such that $\cdot \,|\, \cdot \,|\, \cdot \vdash C[e], C[e'] : \mathsf{int}, \emptyset$, whenever $(C[e] \,|\, h) \downarrow$ then $(C[e'] \,|\, h) \downarrow$.

Here, as usual, $(e \,|\, h) \downarrow$ means that $(e \,|\, h) \longmapsto {}^*(v \,|\, h')$ for some value v and some h'.

Contextual equivalence, $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e \approx e' : \tau, \varepsilon$, is then defined as $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e \lesssim_{\downarrow} e' : \tau, \varepsilon$ and $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e' \lesssim_{\downarrow} e : \tau, \varepsilon$. Note that the diverging behaviours introduced by our operational semantics of atomic e do not influence may-contextual equivalence.

3 Definition of the logical relation

Semantic Types and Worlds We give a Kripke or world-indexed logical relation. This is a fairly standard approach to modeling dynamic allocation; in combination with higher-order store, however, it comes with a fairly standard problem: the *type-world circularity*. Roughly, semantic types are indexed over worlds and worlds contain semantic types, so both need to be defined before the other. A specific instance of this circularity was solved recently by Thamsborg and Birkedal [26] based on metric-space theory developed by Birkedal et. al. [7]; we re-use that solution here. Semantic types (and worlds) are constructed as a fixed-point of a endo-functor on a certain category of metric-spaces. We do not care about that, though; we just give the result of the construction. In addition, we largely ignore the fact that we actually deal in metric spaces and not just plain sets; the little metric machinery we need is deferred to the appendix of the long version of the paper.

There is a set $\mathbf{T}$ of *semantic types* and a set $\mathbf{W}$ of *worlds*; types are world-indexed relations on values and worlds describe the regions and type-layouts of heaps, roughly speaking. Take a type $\mu \in \mathbf{T}$ and apply it to a world $w \in \mathbf{W}$ and you get an indexed relation on values, i.e., $\mu(w) \subseteq \mathbb{N} \times \mathcal{V} \times \mathcal{V}$. These relations are downwards closed in the first coordinate; we read $(k, v_1, v_2) \in \mu(w)$ as saying that v_1 and v_2 are related at type μ up to approximation k assuming world w.

We assume a countably infinite set of region names $\mathcal{RN}$; a world $w \in \mathbf{W}$ contains finitely many such $|w| \subseteq_{\mathit{fin}} \mathcal{RN}$. Some of these $\mathrm{dom}(w) \subseteq |w|$ are *live* and the rest are *dead*. To each live region $r \in \mathrm{dom}(w)$ we associate a finite partial bijection $w(r)$ on locations decorated with types, i.e., $w(r) \subseteq_{\mathit{fin}} \mathcal{L} \times \mathcal{L} \times \widehat{\mathbf{T}}$ such that for $(l_1, l_2, \mu), (m_1, m_2, \nu) \in w(r)$ we have that both $l_1 = m_1$ and $l_2 = m_2$ imply $l_1 = m_1$, $l_2 = m_2$ and $\mu = \nu$. We write $\mathrm{dom}_1(w(r))$ for the set of left hand side locations in the bijection and $\mathrm{dom}_2(w(r))$ for the right hand side ones; different regions must have disjoint left and right hand side locations. For convenience, we set $\mathrm{dom}_1^{\mathrm{A}}(w) = \bigcup_{r \in A \cap \mathrm{dom}(w)} \mathrm{dom}_1(w(r))$ whenever $A \subseteq |w|$, and we write $\mathrm{dom}_1(w)$ for $\mathrm{dom}_1^{|\mathrm{w}|}(w)$, i.e., the set of all left hand side locations. Similarly for the right hand side.

Worlds evolve and types adapt. Triples of two locations and a type can be added to a live region, as long as different regions remain disjoint. Orthogonal to this, one can add a fresh, i.e., neither live nor dead, region name with an associated empty partial bijection. And one can kill any live region, rendering it dead and losing the associated the partial bijection in the process. The reflexive, transitive closure of all three combined is a preorder $\sqsubseteq$ on worlds; it is a crucial property of types that they respect this, i.e., that $w \sqsubseteq w' \implies \mu(w) \subseteq \mu(w')$ for any two $w, w' \in \mathbf{W}$ and any $\mu \in \mathbf{T}$. This is *type monotonicity* and it prevents values from fleeing types over time.

Finally, to tie the knot, there is an isomorphism $\iota : \widehat{\mathbf{T}} \to \mathbf{T}$ from the odd types stored in worlds to proper types. Whenever a type is extracted from a world it needs to be coerced by this isomorphism before it can be applied to some world.

The Logical Relation and Interpretation of Types Often, a logical relation goes like this: two computations are related if they (from related heaps) reduce to related values (and heaps); this is the *extensional* view: we do not care about the intermediate states. As we consider concurrency, however, a computation can be interrupted and so we need to start caring. In our setup, public regions are accessible from the environment. To address this, we assume that before each reduction step, the public regions hold related values; in return, we promise related values after the step. In other words, the *granularity of extensionality* is just one step for the public regions. For private regions, however, there is no interference and the granularity is an entire computation as usual. This is the fundamental idea; it is how we propose to stay extensional in the face of concurrency.

Without further ado, let us look into the cornerstone of our model: the safety relation defined in Figure 6; auxiliary relations are defined in Figure 8. What does it mean to have

$$(k, h_1^\circ, h_2^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\, w^\circ, w?$$

Overall, it says that after environment interference, we can match the behavior of e_1, i.e., termination or any one-step reduction, by zero or more steps of e_2; match in the sense of (re-)establishing certain relations, including safety itself. Safety is a *local* property of a pair of computations, this is crucial: it has no knowledge of computations running concurrently and h_1 and h_2 are the *local heaps*, i.e., the parts of the global heaps that the e_1 respectively e_2 control exclusively. The computations consider $R(\Pi)$ to be their public, $R(\Lambda)$ to be their private and A to be their *anonymous* regions. The latter intuitively are private regions that have been masked out: they exist only for the duration of these computations, but we have to track them to deny the environment access; this is another difficulty imposed by concurrency. Safety is indexed by a world w as well; note that worlds are *global* things: all concurrent threads share one world, i.e., they agree about the division of the heap into regions and the types associated to locations. Finally k is intuitively the number of steps we are safe for, h_1° and h_2° are the (private parts of) the initial local heaps, τ is the expected return type, ε the effects and w° the initial world.

We unroll the definition in writing. The first pair of big square brackets — the *prerequisites* — translates to 'the environment interferes'. This yields a new world w' subject to the constraints of the environment transition relation: no public, private or anonymous regions are killed, and the latter two see no allocation either. The actual contents of the public regions are unknown, but we are free to assume that they hold related values of the proper type, at least where we have read effects; this is the *public heaps* g_1 and g_2 in the precondition relation. In addition we have *frames* f_1 and f_2 that cover the remainder of the world and a triple-split relation that ensures coherence between the domains of corresponding parts of the world and the heaps, see Figures 4 and 8.

The left hand side is irreducible in the *termination branch* and takes one step in the *progress branch*. In either case, we must match this in zero or more steps on the right hand

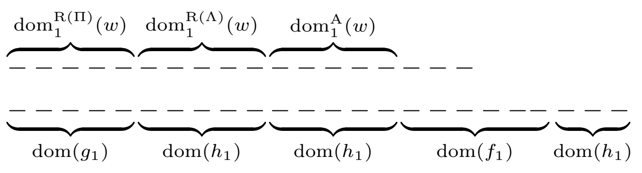
$\mathrm{dom}_1^{\mathrm{R}(\Pi)}(w)$
$\mathrm{dom}_1^{\mathrm{R}(\Lambda)}(w)$
$\mathrm{dom}_1^{\mathrm{A}}(w)$
$\mathrm{dom}(g_1)$
$\mathrm{dom}(h_1)$
$\mathrm{dom}(h_1)$
$\mathrm{dom}(f_1)$
$\mathrm{dom}(h_1)$

Figure 4 The left hand side of the triple-split relation. The top dashed line is $\mathrm{dom}_1(w)$, the bottom dashed line $\mathrm{dom}(g_1 \cdot h_1 \cdot f_1)$. The local heap h_1 has a *private* part matching the private regions, an *anonymous* part matching the anonymous regions and an *off-world* part outside the domain of the world. The frame f_1 must cover regions that are neither public, private nor anonymous.

side, not touching the frame; this means finding a future world w'' and relating a number of things. The choice of future world is restricted by the self transition relation: we must not kill private or public regions, but we can allocate in them, and regions that we know nothing about must be left untouched; this is our promise to the environment. In the termination branch, we are furthermore required to kill off all anonymous regions as the computation is done; any new regions added in the progress branch go to the set of anonymous regions. In both branches, the changes made to the public heap must be well-typed and permitted by the effects and, if we are done, we check the changes made to (the private part of) the local heaps as well; the fact that the public heaps are compared across a single stage and the (private parts of) the local heaps are compared across the entire computations is the crux of the idea of having different granularities of extensionality.

In addition to performing actual allocation, we have the possibility of moving existing locations from, say, the off-world part of the local heap into the public heap or the private part of the local heap; this is a subtle point that permits the actual allocation of new locations and the corresponding extension of the world to be temporarily out of sync.

We have glossed over one aspect of safety: the right hand side takes steps in the ordinary operational semantics, but the left hand side works in the *instrumented operational semantics*. A reduction $(e \,|\, h) \to^n_\mu (e' \,|\, h')$ in the latter implies a similar reduction in former; in addition it counts the steps of a reduction with all atomic commands ‘unfolded’ (with unfolding itself counting one step) and it records all heap accesses; the formal definition is deferred to the appendix of the long version of the paper. We need the former for compatibility of the atomic typing rule below: atomic commands really unfold as they execute, hence we must count the number of ‘unfolded’ steps. It is less immediate that we must test the actual reads, writes and allocations, recorded by μ, against the effects described by ε, as done in the progress branch of safety. But if omitted, our present proof of the Parallelization Theorem falls short, since it relies on the following simple, but crucial commutation property:

▶ **Lemma 2.** *If we have $l \notin \mu$ and $(e \,|\, h) \to^n_\mu (e' \,|\, h')$, then $(e \,|\, h[l \mapsto v]) \to^n_\mu (e' \,|\, h'[l \mapsto v])$.*

The actual logical relation is given in Figure 7. The existentially quantified $a \in \mathbb{N}$ is the minimal number of anonymous regions required to run; apart from that it uses safety in a straightforward way. There is some asymmetry to these definitions: the anonymous regions A are required to exist (and be empty) in the world beforehand, but are killed off in the termination branch; also the precondition on the (private parts of) the initial local heaps is in the logical relation whereas the postcondition lives in the termination branch. The interpretation of types is in Figure 5. Interpreting the function type looks daunting, but a function is just a suspended expression with a single free variable, hence we have to restate most of the logical relation in the definition. Apart from that, we just

$$\llbracket 1 \rrbracket^R w = \{(k, (), ()) \mid k \in \mathbb{N}\} \qquad \llbracket \mathsf{int} \rrbracket^R w = \{(k, n, n) \mid k \in \mathbb{N} \wedge n \in \mathbb{Z}\}$$

$$\llbracket \tau_1 \times \tau_2 \rrbracket^R w = \Big\{(k, (v_{11}, v_{21}), (v_{12}, v_{22})) \mid (k, v_{11}, v_{12}) \in \llbracket \tau_1 \rrbracket^R w \wedge (k, v_{21}, v_{22}) \in \llbracket \tau_2 \rrbracket^R w\Big\}$$

$$\llbracket \mathsf{ref}_\rho \tau \rrbracket^R w = \begin{cases} \left\{ \begin{array}{c} (k, l_1, l_2) \mid \exists \mu \in \widehat{\mathbf{T}}.\, (l_1, l_2, \mu) \in w(R(\rho)) \wedge \\ \forall w' \sqsupseteq w.\ \llbracket \tau \rrbracket^R w' \overset{k}{=} (\iota\, \mu)(w') \end{array} \right\} & R(\rho) \in \mathrm{dom}(w) \\ \{(k, v_1, v_2) \mid k \in \mathbb{N} \wedge v_1, v_2 \in \mathcal{V}\} & R(\rho) \notin \mathrm{dom}(w) \end{cases}$$

$$\llbracket \tau_1 \to_\varepsilon^{\Pi,\Lambda} \tau_2 \rrbracket^R w =$$

$$\begin{cases} \left\{ \begin{array}{l} (k, \mathsf{fun}\, f(x).e_1, \mathsf{fun}\, f(x).e_2) \mid \exists a \in \mathbb{N}.\, \forall j < k.\, \forall w' \sqsupseteq w. \\ \quad \forall A \subseteq \mathrm{dom}(w').\, \forall v_1, v_2 \in \mathcal{V}.\, \forall h_1, h_2, h_1', h_2' \in \mathcal{H}. \\ \left[\begin{array}{l} R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w') \wedge A \# R(\Pi \cup \Lambda) \wedge |A| \geq a \wedge w'(A) = \emptyset \wedge \\ (j, v_1, v_2) \in \llbracket \tau_1 \rrbracket^R w' \wedge h_1' \subseteq h_1 \wedge h_2' \subseteq h_2 \wedge (j, h_1', h_2') \in \mathbf{P}_\varepsilon^{\Lambda,R}\, w' \end{array} \right] \Rightarrow \\ \qquad (j, h_1', h_2', (\mathsf{fun}\, f(x).e_1)\, v_1, (\mathsf{fun}\, f(x).e_2)\, v_2, h_1, h_2) \in \mathbf{safe}_{\tau_2,\varepsilon}^{\Pi,\Lambda,A,R}\, w', w' \end{array} \right\} & \\ & R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w) \\ \{(k, v_1, v_2) \mid k \in \mathbb{N} \wedge v_1, v_2 \in \mathcal{V}\} & R(\mathrm{FRV}(\varepsilon)) \nsubseteq \mathrm{dom}(w) \end{cases}$$

Figure 5 Interpretation of types. We require $R : \mathcal{RV} \rightharpoonup_{\mathit{fin}} \mathcal{RN}$ injective with $\mathrm{FRV}(\tau) \subseteq \mathrm{dom}(R)$. We assume $R(\mathrm{FRV}(\tau)) \subseteq |w|$ above, otherwise we define $\llbracket \tau \rrbracket^R w$ to be the empty set. In the interpretation of functions, and also below, we write $\#$ to denote disjoint sets. We get that $\llbracket \tau \rrbracket^R \in \mathbf{T}$.

remark that the $R(\rho) \notin \mathrm{dom}(w)$ case of reference interpretation is part of an approach to handling dangling pointers (due to region masking) proposed recently in [26]; similarly for the $R(\mathrm{FRV}(\varepsilon)) \nsubseteq \mathrm{dom}(w)$ case.

To conclude this subsection we give a theorem that, combined with the upcoming compatibility, means that logical relatedness implies contextual may-approximation. The proof is in the appendix of the long version of the paper and it is not hard, but it is worth noting that we need a proof at all: with sequential languages, this is a result one reads off the definition of the logical relation.

▶ **Theorem 3** (May-Equivalence). *Assume that* $\cdot \,|\, \cdot \,|\, \cdot \models e_1 \preceq e_2 : \mathsf{int}, \emptyset$ *holds. Take any* $h_1, h_2 \in \mathcal{H}$. *If there are* e_1', h_1' *with* $(e_1 \,|\, h_1) \overset{*}{\longmapsto} (e_1' \,|\, h_1')$ *such that* $\mathrm{irr}(e_1' | h_1')$ *holds, then there is* $n \in \mathbb{Z}$ *such that* $e_1' = \underline{n}$ *and* h_2' *such that* $(e_2 \,|\, h_2) \overset{*}{\longmapsto} (\underline{n}, h_2')$.

Compatibility of the Logical Relation The logical relation is compatible, i.e., respects all typing rules. This is a *sine qua non* of logical relations; it implies the fundamental lemma stating that every well-typed expression is related to itself. And, as discussed just above, it makes the logical relation approximate contextual may-approximation:

▶ **Theorem 4.** $\Pi \,|\, \Lambda \,|\, \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *implies* $\Pi \,|\, \Lambda \,|\, \Gamma \vdash e_1 \lesssim_\downarrow e_2 : \tau, \varepsilon$.

Compatibility means that each typing rule induces a lemma by reading the (unary) typing judgments as the corresponding (binary) logical relations.The three most interesting of these have to do with concurrency and the divide between public and private regions; they are listed here and proofs are given in the appendix of the long version of the paper:

▶ **Lemma 5.** $\Pi \,|\, \Lambda, \rho \,|\, \Gamma \models e_1 \preceq e_2 : \tau,\, \varepsilon$ *implies* $\Pi \,|\, \Lambda \,|\, \Gamma \models e_1 \preceq e_2 : \tau,\, \varepsilon - \rho$ *provided that* $\rho \notin \mathrm{FRV}(\Gamma, \tau)$.

$$
\begin{array}{l}
(k, h_1^\circ, h_2^\circ, e_1, e_2, h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R}\, w^\circ, w \\
\quad \Longleftrightarrow \\
\forall j \leq k.\, \forall w', g_1, g_2, f_1, f_2. \\
\quad \Big[\mathbf{envtran}^{\Pi,\Lambda,A,R}\, w, w' \wedge (j, g_1, g_2) \in \mathbf{P}_{\varepsilon}^{\Pi,R}\, w' \wedge \\
\quad (g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R}\, w'\Big] \Rightarrow \\
\quad \Big[\mathrm{irr}(e_1 | g_1 \cdot h_1 \cdot f_1) \Rightarrow \\
\quad\quad \exists e_2', w'', h_1', h_2', g_1', g_2'. \\
\quad\quad\quad (e_2 \,|\, g_2 \cdot h_2 \cdot f_2) \overset{*}{\longmapsto} (e_2' \,|\, g_2' \cdot h_2' \cdot f_2) \wedge \mathbf{selftran}^{\Pi,\Lambda,A,R}\, w', w'' \wedge \\
\quad\quad\quad \emptyset = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w')) \wedge g_1 \cdot h_1 = g_1' \cdot h_1' \wedge \\
\quad\quad\quad (g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,\emptyset,R}\, w'' \wedge (j, g_1, g_2, g_1', g_2') \in \mathbf{Q}_{\varepsilon}^{\Pi,R}\, w', w'' \wedge \\
\quad\quad\quad (j, e_1, e_2') \in [\![\tau]\!]^R\, (w'') \wedge \exists h_1'' \subseteq h_1', h_2'' \subseteq h_2'.\, (j, h_1^\circ, h_2^\circ, h_1'', h_2'') \in \mathbf{Q}_{\varepsilon}^{\Lambda,R}\, w^\circ, w''\Big] \wedge \\
\quad \Big[\forall e_1', h_1^\dagger, \mu, n \leq j.\, (e_1 \,|\, g_1 \cdot h_1 \cdot f_1) \to_\mu^n (e_1' \,|\, h_1^\dagger) \Rightarrow \\
\quad\quad \exists e_2', w'', A', h_1', h_2', g_1', g_2'. \\
\quad\quad\quad (e_2 \,|\, g_2 \cdot h_2 \cdot f_2) \overset{*}{\longmapsto} (e_2' \,|\, g_2' \cdot h_2' \cdot f_2) \wedge \mathbf{selftran}^{\Pi,\Lambda,A,R}\, w', w'' \wedge \\
\quad\quad\quad A' = (A \cap \mathrm{dom}(w'')) \cup (\mathrm{dom}(w'') \setminus \mathrm{dom}(w')) \wedge h_1^\dagger = g_1' \cdot h_1' \cdot f_1 \wedge \\
\quad\quad\quad (g_1', h_1', f_1, g_2', h_2', f_2) \in \mathbf{splits}^{\Pi,\Lambda,A',R}\, w'' \wedge (j-n, g_1, g_2, g_1', g_2') \in \mathbf{Q}_{\varepsilon}^{\Pi,R}\, w', w'' \wedge \\
\quad\quad\quad \mu \in \mathbf{effs}_{\varepsilon,h_1'}^{A',R}\, w'' \wedge (j-n, h_1^\circ, h_2^\circ, e_1', e_2', h_1', h_2') \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A',R}\, w^\circ, w''\Big]
\end{array}
$$

Figure 6 Safety. The predicate is defined by well-founded induction. Nontrivial requirements are: $\Pi \# \Lambda$, $\mathrm{FRV}(\tau, \varepsilon) \subseteq \Pi \cup \Lambda$, $\mathrm{FV}(e_1, e_2) = \emptyset$, $R : \Pi \cup \Lambda \hookrightarrow |w^\circ|$, $R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w^\circ)$ and $w \sqsupseteq w^\circ$ with $\mathrm{dom}(w^\circ) \cap R(\Pi \cup \Lambda) \subseteq \mathrm{dom}(w)$, $A \subseteq \mathrm{dom}(w)$ and $A \# R(\Pi \cup \Lambda)$. See Figure 8 for auxiliary definitions. We refer to the contents of the big square brackets as the *prerequisites*, the *termination branch* and the *progress branch*, respectively.

$$
\begin{array}{l}
\Pi \mid \Lambda \mid \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon \\
\quad \Longleftrightarrow \\
\exists a \in \mathbb{N}.\, \forall k \in \mathbb{N}.\, \forall w \in \mathbf{W}.\, \forall R : \Pi \cup \Lambda \hookrightarrow |w|.\, \forall A \subseteq \mathrm{dom}(w). \\
\ \forall \gamma_1, \gamma_2 \in \mathcal{V}^{|\Gamma|}.\, \forall h_1, h_2, h_1', h_2' \in \mathcal{H}. \\
\quad \Big[R(\mathrm{FRV}(\varepsilon)) \subseteq \mathrm{dom}(w) \wedge A \# R(\Pi \cup \Lambda) \wedge |A| \geq a \wedge \forall r \in A.\, w(r) = \emptyset \wedge \\
\quad (k, \gamma_1, \gamma_2) \in [\![\Gamma]\!]^R w \wedge h_1' \subseteq h_1 \wedge h_2' \subseteq h_2 \wedge (k, h_1', h_2') \in \mathbf{P}_{\varepsilon}^{\Lambda,R} w\Big] \Rightarrow \\
\quad\quad (k, h_1', h_2', e_1[\gamma_1/\Gamma], e_2[\gamma_2/\Gamma], h_1, h_2) \in \mathbf{safe}_{\tau,\varepsilon}^{\Pi,\Lambda,A,R} w, w.
\end{array}
$$

Figure 7 The logical relation with anonymous regions. We require that $\Pi \# \Lambda$, $\mathrm{FRV}(\Gamma, \tau, \varepsilon) \subseteq \Pi \cup \Lambda$ and, as always, that $\mathrm{FV}(e_1, e_2) \in |\Gamma|$.

$$\begin{aligned}
&\mathbf{envtran}^{\Pi,\Lambda,A,R}\, w, w' \iff w \sqsubseteq w' \wedge \forall r \in \mathrm{dom}(w) \cap (R(\Pi \cup \Lambda) \cup A).\, r \in \mathrm{dom}(w') \\
&\quad \wedge \forall r \in \mathrm{dom}(w) \cap (R(\Lambda) \cup A).\, w(r) = w'(r).
\end{aligned}$$

$$\begin{aligned}
&\mathbf{selftran}^{\Pi,\Lambda,A,R}\, w, w' \iff w \sqsubseteq w' \wedge \forall r \in \mathrm{dom}(w) \setminus A.\, r \in \mathrm{dom}(w') \\
&\quad \wedge \forall r \in \mathrm{dom}(w) \setminus (R(\Pi \cup \Lambda) \cup A).\, w(r) = w'(r).
\end{aligned}$$

$$\begin{aligned}
&(g_1, h_1, f_1, g_2, h_2, f_2) \in \mathbf{splits}^{\Pi,\Lambda,A,R}\, w \iff \\
&\quad \mathrm{dom}(h_1) \# \mathrm{dom}(g_1) \# \mathrm{dom}(f_1) \wedge \mathrm{dom}(h_2) \# \mathrm{dom}(g_2) \# \mathrm{dom}(f_2) \wedge \\
&\quad \mathrm{dom}_1^{\mathrm{R}(\Pi)}(w) = \mathrm{dom}(g_1) \wedge \mathrm{dom}_1^{\mathrm{R}(\Lambda)\cup\mathrm{A}}(w) \subseteq \mathrm{dom}(h_1) \wedge \\
&\quad \mathrm{dom}_1^{\mathrm{dom(w)}\setminus(\mathrm{R}(\Pi\cup\Lambda)\cup\mathrm{A})}(w) \subseteq \mathrm{dom}(f_1) \wedge \\
&\quad \mathrm{dom}_2^{\mathrm{R}(\Pi)}(w) = \mathrm{dom}(g_2) \wedge \mathrm{dom}_2^{\mathrm{R}(\Lambda)\cup\mathrm{A}}(w) \subseteq \mathrm{dom}(h_2) \wedge \\
&\quad \mathrm{dom}_2^{\mathrm{dom(w)}\setminus(\mathrm{R}(\Pi\cup\Lambda)\cup\mathrm{A}))}(w) \subseteq \mathrm{dom}(f_2).
\end{aligned}$$

$$\begin{aligned}
&(k, h_1, h_2) \in \mathbf{P}_\varepsilon^{\Theta,R}\, w \iff \mathrm{dom}(h_1) = \mathrm{dom}_1^{\mathrm{R}(\Theta)}(w) \wedge \mathrm{dom}(h_2) = \mathrm{dom}_2^{\mathrm{R}(\Theta)}(w) \wedge \\
&\quad \forall r \in R(\Theta) \cap \mathrm{dom}(w).\, \forall (l_1, l_2, \mu) \in w(r). \\
&\quad r \in R(\mathrm{rds}\,\varepsilon) \Rightarrow k > 0 \Rightarrow (k-1, h_1(l_1), h_2(l_2)) \in (\iota\,\mu)(w).
\end{aligned}$$

$$\begin{aligned}
&(k, h_1, h_2, h_1', h_2') \in \mathbf{Q}_\varepsilon^{\Theta,R}\, w, w' \iff \\
&\quad \mathrm{dom}(h_1) = \mathrm{dom}_1^{\mathrm{R}(\Theta)}(w) \wedge \mathrm{dom}(h_2) = \mathrm{dom}_2^{\mathrm{R}(\Theta)}(w) \wedge \\
&\quad \mathrm{dom}(h_1') = \mathrm{dom}_1^{\mathrm{R}(\Theta)}(w') \wedge \mathrm{dom}(h_2') = \mathrm{dom}_2^{\mathrm{R}(\Theta)}(w') \wedge \\
&\quad \big(\forall r \in R(\Theta) \cap \mathrm{dom}(w).\, \forall (l_1, l_2, \mu) \in w(r). \\
&\qquad [h_1(l_1) = h_1'(l_1) \wedge h_2(l_2) = h_2'(l_2)] \vee [r \in R(\mathrm{wrs}\,\varepsilon) \wedge \\
&\qquad\; k > 0 \Rightarrow (k-1, h_1'(l_1), h_2'(l_2)) \in (\iota\,\mu)(w')]\big) \wedge \\
&\quad \big(\forall r \in R(\Theta) \cap \mathrm{dom}(w). \\
&\qquad \forall (l_1, l_2, \mu) \in w'(r) \setminus w(r).\, r \in R(\mathrm{als}\,\varepsilon) \wedge \\
&\qquad k > 0 \Rightarrow (k-1, h_1'(l_1), h_2'(l_2)) \in (\iota\,\mu)(w')\big).
\end{aligned}$$

$$\begin{aligned}
&\mu \in \mathbf{effs}_{\varepsilon,h}^{A,R}\, w \iff \{l \mid rd_l \in \mu\} \cap \mathrm{dom}_1(w) \subseteq \mathrm{dom}_1^{\mathrm{R}(\mathrm{rds}\,\varepsilon)\cup\mathrm{A}}(w) \wedge \\
&\quad \{l \mid wr_l \in \mu\} \cap \mathrm{dom}_1(w) \subseteq \mathrm{dom}_1^{\mathrm{R}(\mathrm{wrs}\,\varepsilon)\cup\mathrm{A}}(w) \wedge \\
&\quad \{l \mid al_l \in \mu\} \cap \mathrm{dom}_1(w) \subseteq \mathrm{dom}_1^{\mathrm{R}(\mathrm{als}\,\varepsilon)\cup\mathrm{A}}(w) \wedge \\
&\quad \{l \mid rd_l \in \mu \vee wr_l \in \mu \vee al_l \in \mu\} \setminus \mathrm{dom}_1(w) \subseteq \mathrm{dom}(h).
\end{aligned}$$

Figure 8 Six auxiliary definitions. The *environment transition* and *self transition* relations are defined for $\Pi \# \Lambda$, $R : \Pi \cup \Lambda \hookrightarrow |w|$, $A \subseteq \mathrm{dom}(w)$ and $R(\Pi \cup \Lambda) \# A$. The *triple-split* relation has the same prerequisites. The *precondition* relation is defined for $R : \mathcal{RV} \rightharpoonup_{fin} |w|$ injective with $\Theta \cup \mathrm{FRV}(\varepsilon) \subseteq \mathrm{dom}(R)$. The *postcondition* relation additionally requires $w' \sqsupseteq w$ such that $\mathrm{dom}(w) \cap R(\Theta) \subseteq \mathrm{dom}(w')$. Finally the *actual-effects* relation expects $R : \mathcal{RV} \rightharpoonup_{fin} |w|$ injective with $\mathrm{FRV}(\varepsilon) \subseteq \mathrm{dom}(R)$ and $A \subseteq \mathrm{dom}(w)$.

▶ **Lemma 6.** $\cdot \,|\, \Pi, \Lambda \,|\, \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *implies* $\Pi \,|\, \Lambda \,|\, \Gamma \models \mathsf{atomic}\ e_1 \preceq \mathsf{atomic}\ e_2 : \tau, \varepsilon$ *if* $\mathrm{als}\,\varepsilon \subseteq \mathrm{rds}\,\varepsilon \cap \mathrm{wrs}\,\varepsilon$.

▶ **Lemma 7.** $\Pi, \Lambda \,|\, \cdot \,|\, \Gamma \models e_1 \preceq e_2 : \tau, \varepsilon$ *and* $\Pi, \Lambda \,|\, \cdot \,|\, \Gamma \models e_1^\dagger \preceq e_2^\dagger : \tau^\dagger, \varepsilon^\dagger$ *together imply* $\Pi \,|\, \Lambda \,|\, \Gamma \models \mathsf{par}\ e_1\ \mathsf{and}\ e_1^\dagger \preceq \mathsf{par}\ e_2\ \mathsf{and}\ e_2^\dagger : \tau \times \tau^\dagger, \varepsilon \cup \varepsilon^\dagger$.

4 Applications

4.1 Parallelization Theorem: Disjoint Concurrency

We now explain our Parallelization Theorem, which gives us an easy way to prove properties about the common case of disjoint concurrency, where disjointness is captured using private regions and effect annotations.

▶ **Theorem 8** (Parallelization). *Assuming that*

1. $\Pi, \Lambda \mid \cdot \mid \Gamma \vdash e_1 : \tau_1, \varepsilon_1$,
2. $\Pi, \Lambda \mid \cdot \mid \Gamma \vdash e_2 : \tau_2, \varepsilon_2$,
3. $\mathrm{rds}\,\varepsilon_1 \cup \mathrm{wrs}\,\varepsilon_1 \cup \mathrm{rds}\,\varepsilon_2 \cup \mathrm{wrs}\,\varepsilon_2 \subseteq \Lambda$,
4. $\mathrm{rds}\,\varepsilon_1 \cap \mathrm{wrs}\,\varepsilon_2 = \mathrm{rds}\,\varepsilon_2 \cap (\mathrm{wrs}\,\varepsilon_1 \cup \mathrm{als}\,\varepsilon_1) = \mathrm{wrs}\,\varepsilon_1 \cap \mathrm{wrs}\,\varepsilon_2 = \emptyset$,

the following property holds:

$$\Pi \mid \Lambda \mid \Gamma \models \langle e_1, e_2 \rangle \cong \mathsf{par}\ e_1\ \mathsf{and}\ e_2 : \tau_1 \times \tau_2, \varepsilon_1 \cup \varepsilon_2.$$

Intuitively, item 3 keeps the environment from detecting anything, and item 4 prevents the two computations from talking among themselves, thereby making them independent; the $\mathrm{als}\,\varepsilon_1$ in item 4 is a technicality that we cannot do without. We showed a concrete simple application of this theorem in the Introduction. More generally, example usage includes situations where we operate on two imperative data structures (say linked lists or graphs); if we only mutate parts of the data structures that are in different regions, then we may safely parallelize operations on the data structures.

The masking rule makes it possible to do more optimizations via the Parallelization Theorem: Consider, for simplicity, the familiar example of an efficient implementation *fib* of the Fibonacci function using two local references. We can use the masking rule to give it type and effect $\mathsf{int} \rightarrow_{\emptyset}^{\cdot;\cdot} \mathsf{int}, \emptyset$. This allows us to view the imperative implementation as pure, and thus by Theorem 8 we find that it is sound to optimize two sequential calls to *fib* to two parallel calls. This may sound like a simple optimization, but the point is that a compiler can perform it automatically, just based on the effect types. It also underlines how we are able to reason about more involved behaviors of concurrent threads, even though the type system provides only rough bounds on interference through the private-public distinction.

The proof of the Parallelization Theorem is quite tricky. Please see the the appendix of the long version of the paper for an informal overview of the proof and the technical details.

4.2 Non-disjoint Concurrency

We now exemplify how our logical relations model can also be used to reason compositionally about equivalences of fine-grained concurrent programs operating on public regions.

Consider the following type

$$\tau \equiv \mathsf{ref}_\rho \mathsf{int} \rightarrow_{\{rd_\rho, wr_\rho\}}^{\rho,\emptyset} 1$$

of functions that take an integer reference in a public region, possibly read and write from the reference, and return unit. The following two functions

$$\mathsf{fun\,inc}_1(x).\ \mathsf{let}\ y = !\,x\ \mathsf{in\ let}\ z = y+1\ \mathsf{in}\ \mathsf{if\ cas}\ (x,y,z)\ \mathsf{then}\ \langle\rangle\ \mathsf{else\ inc}_1\ (x) \quad \text{and} \quad \mathsf{fun\,inc}_2(x).\ \mathsf{atomic}\ (x := !\,x+1)$$

both have type τ. (We have allowed ourselves to use a standard conditional expression; 1 corresponds to true and 0 to false.) Both functions increment the integer given in their reference arguments; inc_1 uses the fine-grained compare-and-swap to do it atomically, whereas inc_2 uses the brute-force atomic operation. Using our logical relations model, we can prove that inc_1 and inc_2 are contextually equivalent:

$$\rho \mid \cdot \mid \cdot \vdash \mathsf{inc}_1 \approx \mathsf{inc}_2 : \tau, \emptyset. \tag{1}$$

Hence, replacing inc_2 with inc_1 in any well-typed client gives two contextually equivalent expressions. Thus our logical relation models a form of *data abstraction for concurrency* (where we abstract over the granularity of concurrency in the module).

We now show how to use the equivalence of inc_1 and inc_2 to derive equivalences of two different clients using the fine-grained concurrency implementation inc_1.

To this end, consider the following two client programs of type

$$\sigma \equiv \tau \to^{\rho,\emptyset}_{\emptyset} \mathsf{ref}_\rho\mathsf{int} \to^{\rho,\emptyset}_{\{rd_\rho, wr_\rho\}} \mathsf{int},$$

$$\mathsf{fun\,c}_1(\mathsf{inc}).\lambda\,n.\mathsf{inc}\,n;\ \mathsf{inc}\,n;\ !\,n \quad \text{and} \quad \mathsf{fun\,c}_2(\mathsf{inc}).\lambda\,n.(\mathsf{par\ inc}\,n\ \mathsf{and\ inc}\,n);\ !\,n$$

Note that c_1 makes two sequential calls to inc, whereas c_2 runs the two calls in parallel. Because of the use of compare-and-swap in inc_1, we would hope that the $\mathsf{c}_1\,\mathsf{inc}_1$ and $\mathsf{c}_2\,\mathsf{inc}_1$ are contextually equivalent (in typing context $\rho \,|\, \emptyset \,|\, \emptyset$). We can prove that this is indeed the case using compositional reasoning as follows. Using our logical relation, we prove that $\mathsf{c}_1\,\mathsf{inc}_2$ is contextually equivalent to $\mathsf{c}_2\,\mathsf{inc}_2$, i.e.,

$$\rho \mid \cdot \mid \cdot \vdash \mathsf{c}_1\,\mathsf{inc}_2 \approx \mathsf{c}_2\,\mathsf{inc}_2 : \mathsf{ref}_\rho\mathsf{int} \to^{\rho,\emptyset}_{\{rd_\rho, wr_\rho\}} \mathsf{int}, \emptyset. \tag{2}$$

Finally, we conclude that $\mathsf{c}_1\,\mathsf{inc}_1$ is contextually equivalent to $\mathsf{c}_2\,\mathsf{inc}_1$ by transitivity of contextual equivalence (using (1), (2) and (1) again for the respective steps):

$$\mathsf{c}_1\,\mathsf{inc}_1 \approx \mathsf{c}_1\,\mathsf{inc}_2 \approx \mathsf{c}_2\,\mathsf{inc}_2 \approx \mathsf{c}_2\,\mathsf{inc}_1$$

This proof illustrates an important point: to show equivalence of two clients of a module implemented using fine-grained concurrency, it suffices to show that the clients are equivalent wrt. a coarse-grained implementation, and that the coarse-grained implementation is equivalent to the fine-grained implementation. This is often a lot simpler than trying to show the equivalence of the clients wrt. the fine-grained implementation directly. We can think of the coarse-grained implementation of the module (here inc_2) as the *specification* of the module and the fine-grained implementation (here inc_1) as its *implementation*.

The formal proofs of (1) and (2) follow by straightforward induction.

5 Discussion

Gifford and Lucassen [15, 20] originally proposed type-and-effect systems as a static analysis for determining which parts of a higher-order imperative program could be implemented using parallelism. Here we are able to express the formal correctness of these ideas in a succinct way by having a parallel construct in our programming language and establishing the Parallelization Theorem.

In Section 4.2 we showed how contextual equivalence can be used to *state* that compare-and-swap can be used to implement a simple form of locking, and how our logical relations model could be used to *prove* this statement. We believe that it should be possible to give similar succinct statements and proofs of other implementations of synchronization; confer work by Turon and Wand [28].

As mentioned earlier, we have deliberately used the same definition of worlds here as in [26]. As discussed there [26, Section 8.2], this notion of world has somewhat limited expressiveness: the only heap invariants we can state are those that relate values at two locations by a semantic type. To increase expressiveness, it would thus be interesting to extend our model using ideas from [12], and then investigate more examples of equivalences.

Recently, Liang et. al. [19] have proposed RGSim, a simulation based on rely-guarantee, to verify program transformations in a concurrent setting. Their actual definition [19, Definition 4] bears some resemblance to our safety relation; indeed, an early draft of *loc.cit.* was a source of inspiration. They have no division of the heap into public and private parts, instead they give a pair of rely and guarantee that, respectively, constrain the interference of the environment and the actions of the computation. Their approach is essentially untyped; one point of view is that we 'auto-instantiate' the many parameters of their simulation based on our typing information. They consider first-order languages with ground store; this obviously keeps life simple, but the example equivalences they give are not.

Our simple example of data abstraction for concurrency in Section 4.2 suggests that there could be a relationship to linearizability. We intend to explore whether a formal relationship can be established in our higher-order setting; confer work by Filipović et. al. [14].

6 Conclusion and Future Work

We have presented a logical relations model of a new type-and-effect system for a concurrent higher-order ML-like language with general references. We have shown how to use the model for reasoning about both disjoint and non-disjoint concurrency. In particular, we have proved the first automatic Parallelization Theorem for such a rich language.

In this paper, we have focused on may contextual equivalence. Future work includes investigating models for must contextual equivalence. Since our language allows the encoding of countable nondeterminism, must equivalence is non-trivial, and will probably involve indexing over ω_1 rather than ω [24]. Future work also includes extending the model to region and effect polymorphism, as well as the extension to more expressive worlds, and to other concurrency constructs such as fork-join.

The authors would like to thank Jan Schwinghammer and Xinyu Feng for discussions of aspects of this work.

References

1 Amal Ahmed. *Semantics of Types for Mutable State.* PhD thesis, Princeton University, 2004.

2 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *POPL*, 2009.

3 N. Benton, L. Beringer, M. Hofmann, and A. Kennedy. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP*. ACM, 2007.

4 N. Benton, L. Beringer, M. Hofmann, and A. Kennedy. Relational semantics for effect-based program transformations: Higher-order store. In *PPDP*. ACM, 2009.

5 N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI*, 2007.

6 N. Benton, A. Kenney, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *APLAS*, 2006.

7 L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, 2011.

8 L. Birkedal, J. Thamsborg, and K. Støvring. Realizability semantics of parametric polymorphism, general references, and recursive types. In *FOSSACS*, 2009.

9 L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *POPL*, 1996.

10 A. Buisse, L. Birkedal, and K. Støvring. A step-indexed Kripke model of separation logic for storable locks. In *MFPS*, 2011.

11 M. Dodds, X. Feng, M.J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

12 D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *ICFP 2010*, pages 143–156. ACM, 2010.

13 X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, 2007.

14 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *TCS*, 2010.

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

16 F. Henglein, H. Makholm, and H. Niss. Effect types and region-based memory management. In B.C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. 2005.

17 Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *POPL*, 2006.

18 James Laird. A fully abstract trace semantics for general references. In *ICALP*, 2007.

19 H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

20 J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *POPL*, 1988.

21 A. Murawski and N. Tzevelekos. Game semantics for good general references. In *LICS*, 2011.

22 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *TCS*, 2007.

23 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *TOPLAS*, 2011.

24 Jan Schwinghammer and Lars Birkedal. Step-indexed relational reasoning for countable nondeterminism. In *CSL*, 2011.

25 Eijiro Sumii. A complete characterization of observational equivalence in polymorphic λ-calculus with general references. In *CSL*, 2009.

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

27 M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of POPL*, 1994.

28 Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 247–258. ACM, 2011.

29 V. Vafeiadis and M.J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, 2007.

30 Viktor Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, 2011.