Mechanized Relational Verification of Concurrent Programs with Continuations

AMIN TIMANY, imec-Distrinet, KU Leuven, Belgium
LARS BIRKEDAL, Aarhus University, Denmark

Concurrent higher-order imperative programming languages with continuations are very flexible and allow for the implementation of sophisticated programming patterns. For instance, it is well known that continuations can be used to implement cooperative concurrency. Researchers and practitioners have also advocated that the implementation of web servers can be simplified by using a programming pattern based on continuations. This programming pattern can, in particular, help simplify keeping track of the state of clients interacting with the server. However, such advanced programming languages are very challenging to reason about.

In this paper we present the first completely formalized tool for interactive mechanized relational verification of programs written in a concurrent higher-order imperative programming language with continuations (call/cc and throw). In more detail, we develop novel logical relations which can be used to give mechanized proofs of relational properties. In particular, we prove correctness of an implementation of cooperative concurrency with continuations. In addition, we show that that a rudimentary web server implemented using the continuation-based pattern is contextually equivalent to one implemented without the continuation-based pattern.

Additional Key Words and Phrases: Logical relations, Continuations, Concurrency

1 INTRODUCTION

In a programming language with continuations, a computation can be suspended into a continuation object which can be resumed later. Continuations enable interesting programming patterns. For instance, it is well-known that they can be used to implement cooperative concurrency [Haynes et al. 1984]: switching between threads can be implemented by suspending the running thread, storing the suspension and running another thread. Another notable application of continuations is the implementation of continuation-based web servers [Flatt 2017; Krishnamurthi et al. 2007; Queinnec 2004]. Web servers store the state of their communication with each client in order to provide a coherent experience for returning clients. For this purpose continuation-based web servers store the continuation of the server-side program. This helps simplify the web server program because the only thing that the server needs to do in order to serve a returning client appropriately is to resume the continuation corresponding to the last communication with the client.

Both of the aforementioned programming patters involve concurrency, higher-order and imperative aspects of the programming language in combination with continuations in sophisticated and interesting ways. Such expressive and advanced programming languages and programs written in them are known to be challenging to model and reason about. In this paper we present novel techniques for relational reasoning about such programming languages and programs written in them.

Specifically, we develop a new logical relations model for proving contextual refinement of programs written in $\mathsf{F}^{\mu,\,ref}_{conc,\,cc}$, a call-by-value programming language featuring concurrency, impredicative polymorphism, recursive types, dynamically allocated higher-order store and first-class

Authors' addresses: Amin Timany, Department of Computer Science, imec-Distrinet, KU Leuven, Leuven, Belgium, amin.timany@cs.kuleuven.be; Lars Birkedal, Department of Computer Science, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

continuations with call/cc and throw primitives. We employ this logical relations model to prove (1) contextual equivalence of two simple web server implementations: a continuation-based one and a state-storing one; and (2) that one-shot continuations (continuations that can only be used once) can be used to simulate ordinary continuations [Friedman and Haynes 1985]. The latter is a well-known result for sequential programs, see, e.g., Dreyer et al. [2012]. Here we show that it also holds in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, that is, in the presence of concurrency.

In addition, we develop a relational model for showing the correctness of a continuation-based implementation of cooperative concurrency. We consider two programming languages: a language, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$, with built-in cooperative concurrency and a sequential language, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$, featuring continuations. We develop a cross-language logical relation between $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ and use it to show correctness of a translation of cooperative concurrency into one based on continuations.

We define our logical relations models in a variant of the Iris program logic framework [Jung et al. 2016, 2015; Krebbers et al. 2017a]. Iris is a framework for state-of-the-art higher-order concurrent separation logics. We use Iris because (1) it allows us to define our logical relations and reason about them at a higher level of abstraction compared to an explicit model construction; (2) we side-step the well-known type-world-circularity problems [Ahmed 2004; Ahmed et al. 2002; Birkedal et al. 2011] involved in defining logical relations for programming languages with higher-order store (since that is already "taken care of" by the model of Iris); and (3) we can leverage the Coq implementation of the Iris base logic [Krebbers et al. 2017a] and the Iris Proof Mode [Krebbers et al. 2017b] when mechanizing our development in Coq. Indeed, accompanying this paper is a tool for mechanized relational verification of concurrent programs with continuations. The mechanization has been done in Coq and all the results in the paper have been formally verified.

One of the most important features of concurrent separation logics for reasoning about concurrent imperative programs, *e.g.* da Rocha Pinto et al. [2014]; Dinsdale-Young et al. [2013, 2010]; Jung et al. [2016, 2015]; Krebbers et al. [2017a,b]; Ley-Wild and Nanevski [2013]; Nanevski et al. [2014]; O'Hearn [2007]; Sergey et al. [2015]; Svendsen and Birkedal [2014]; Turon et al. [2013a], is the support for *modular* / *local* reasoning. In particular, weakest preconditions and Hoare-triples enable *thread-local* and *context-local* reasoning. Here thread-local means that we can reason about each thread in isolation: when we reason about a particular thread, we need not explicitly consider interactions from other concurrently executing threads. Similarly, context-local means that when we reason about a particular expression, we need not consider under which evaluation context it is being evaluated. The latter is sometimes codified by the soundness of a proof rule such as the following:

$$\textsc{Hoare-Bind (inadmissible in presence of continuations)}\quad \frac{\{P\}\, e\, \{\Psi\} \qquad \forall w.\ \{\Psi(w)\}\, K[w]\, \{\Phi\}}{\{P\}\, K[e]\, \{\Phi\}}$$

The Hoare-triple $\{P\}\, e\, \{\Psi\}$ intuitively means that, given precondition P, expression e is *safe* and, whenever it reduces to a value v, we are guaranteed that $\Psi(v)$ holds. Intuitively, the above rule expresses that to prove a Hoare triple for an expression e in an evaluation context K, it suffices to prove a property for e in isolation from K, and then show that the desired postcondition Φ can be obtained when substituting a value w satisfying the postcondition Ψ for e into the evaluation context. In a programming language with control operators, *e.g.* call/cc and throw, the context under which a program is being evaluated is of utmost importance, and thus the above proof rule is *not* sound in general.

Thus, in general, when reasoning about concurrent programs with continuations in a concurrent separation logic, we cannot use context-local reasoning. Hence as part of this work, we develop new

non-context-local proof rules for Hoare triples. Those are somewhat more elaborate to use than the standard context-local rules, but that is the price we have to pay to be able to reason in general about non-local control flow. We define our logical relation in terms of Hoare triples, following earlier work [Krebbers et al. 2017b; Turon et al. 2013a], and thus we use the non-context-local proof rules for establishing contextual equivalence of concurrent programs with continuations (and also when proving the soundness of the logical relation itself). To simplify reasoning about parts of programs that do not use control operators, we introduce a new notion of *context-local Hoare triples*. They are defined in terms of the non-context-local Hoare triples and therefore we are able to mix and match reasoning steps using (non-context local) Hoare triples and context-local Hoare triples.

Contributions. In this paper, we make the following contributions:

- We present a program logic (weakest preconditions and Hoare-triples) for reasoning about programs written in $\mathsf{F}^{\mu,\,ref}_{conc,\,cc}$, a programming language with impredicative polymorphism, recursive types, higher-order functions, higher-order store, concurrency and first-class continuations.
- We present context-local weakest-preconditions and Hoare-triples which simplify reasoning about programs without non-local control flow.
- We present a novel logical relations model for $\mathsf{F}^{\mu,\,ref}_{conc,\,cc}$.
- We use our logical relations model and context-local reasoning to prove equivalence of two simple web server implementations: a continuation-based one and a state-storing one.
- We further use our logical relations model to prove correctness of Friedman and Haynes [1985] encoding of continuations by means of one-shot continuations in a concurrent programming language.
- We develop a cross-language logical relations model between $\mathsf{F}^{\mu,\,ref}_{cc}$ and $\mathsf{F}^{\mu,\,ref}_{cc,\,coop}$ for proving program refinement.
- We use our cross-language logical relations model to prove correctness of a continuation-based implementation of cooperative concurrency.
- We have developed a fully formalized tool for mechanized interactive relational verification of concurrent programs with continuations. Our tool is developed on top of Iris, a state-of-the-art program logic framework, and we have used it to mechanize all of our contributions in the Coq proof assistant.

2 THE LANGUAGE: $\mathsf{F}^{\mu,\,ref}_{conc,\,cc}$

The language that we consider in this paper, $\mathsf{F}^{\mu,\,ref}_{conc,\,cc}$, is a typed lambda calculus with a standard call-by-value small-step operational semantics. It features impredicative polymorphism, recursive types, higher-order mutable references, fine-grained concurrency and first-class continuations. The types of $\mathsf{F}^{\mu,\,ref}_{conc,\,cc}$ are as follows:

$$\tau ::= \alpha \mid 1 \mid \mathbb{B} \mid \mathbb{N} \mid \tau \to \tau \mid \forall \alpha.\, \tau \mid \mu \alpha.\, \tau \mid \tau \times \tau \mid \tau + \tau \mid \mathsf{ref}(\tau) \mid \mathsf{cont}(\tau)$$

The type $\mathsf{ref}(\tau)$ is the type of references with contents of type τ and $\mathsf{cont}(\tau)$ is the type of continuations that can be resumed by throwing them a value of type τ.

The syntax for expressions and values is:

$$\begin{aligned} e ::= {} & x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid e \odot e \mid \mathsf{rec}\, f(x) = e \mid e\ e \mid \Lambda\, e \mid e\ \_ \mid \mathsf{fold}\, e \mid \mathsf{unfold}\, e \mid (e, e) \mid \pi_i\, e \\ & \mid \mathsf{inj}_i\, e \mid \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_i\, x \Rightarrow e_i\, \mathsf{end} \mid \ell \mid \mathsf{ref}(e) \mid\ !\, e \mid e \leftarrow e \mid \mathsf{cas}(e, e, e) \mid \mathsf{fork}\ \{e\} \\ & \mid \mathsf{cont}(K) \mid \mathsf{call/cc}\ (x.\, e) \mid \mathsf{throw}\, e\, \mathsf{to}\, e \end{aligned}$$

$$\textsc{T-Var}\ \frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau} \qquad \textsc{T-Unit}\ \Xi \mid \Gamma \vdash () : 1 \qquad \textsc{T-Nat}\ \Xi \mid \Gamma \vdash n : \mathbb{N} \qquad \textsc{T-TLam}\ \frac{\Xi, \alpha \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall \alpha.\, \tau}$$

$$\textsc{T-Rec}\ \frac{\Xi \mid \Gamma, x : \tau, f : \tau \to \tau' \vdash e : \tau'}{\Xi \mid \Gamma \vdash \ \mid \mathrm{rec}\, f(x) = e : \tau \to \tau'} \qquad \textsc{T-TApp}\ \frac{\Xi \mid \Gamma \vdash e : \forall \alpha.\, \tau}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/\alpha]} \qquad \textsc{T-Fold}\ \frac{\Xi \mid \Gamma \vdash e : \tau[\mu\alpha.\, \tau/\alpha]}{\Xi \mid \Gamma \vdash \mathrm{fold}\, e : \mu\alpha.\, \tau}$$

$$\textsc{T-UnFold}\ \frac{\Xi \mid \Gamma \vdash e : \mu\alpha.\, \tau}{\Xi \mid \Gamma \vdash \mathrm{unfold}\, e : \tau[\mu\alpha.\, \tau/\alpha]} \qquad \textsc{T-Ref}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathrm{ref}(e) : \mathrm{ref}(\tau)} \qquad \textsc{T-DeRef}\ \frac{\Xi \mid \Gamma \vdash e : \mathrm{ref}(\tau)}{\Xi \mid \Gamma \vdash\, !\, e : \tau}$$

$$\textsc{T-Fork}\ \frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathrm{fork}\ \{e\} : 1} \qquad \textsc{T-Cont}\ \frac{K : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi \mid \Gamma; \tau')}{\Xi \mid \Gamma \vdash \mathrm{cont}(K) : \mathrm{cont}(\tau)} \qquad \textsc{T-Assign}\ \frac{\Xi \mid \Gamma \vdash e : \mathrm{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : 1}$$

$$\textsc{T-CAS}\ \frac{\Xi \mid \Gamma \vdash e_1 : \mathrm{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau \qquad \Xi \mid \Gamma \vdash e_3 : \tau}{\Xi \mid \Gamma \vdash \mathrm{cas}(e_1, e_2, e_3) : 1} \qquad \textsc{T-Call/cc}\ \frac{\Xi \mid \Gamma, x : \mathrm{cont}(\tau) \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathrm{call/cc}\,(x.\, e) : \tau}$$

$$\textsc{T-Throw}\ \frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \mid \Gamma \vdash e' : \mathrm{cont}(\tau)}{\Xi \mid \Gamma \vdash \mathrm{throw}\ e\ \mathrm{to}\ e' : \tau'}$$

Fig. 1. An excerpt of the typing rules

$$v ::= () \mid \mathrm{true} \mid \mathrm{false} \mid n \mid \mathrm{rec}\, f(x) = e \mid \Lambda\, e \mid \mathrm{fold}\, v \mid (v, v) \mid \mathrm{inj}_i\, v \mid \ell \mid \mathrm{cont}(K)$$

We write n for natural numbers and the symbol $\circledcirc$ stands for binary operations on natural numbers (both basic arithmetic operations and basic comparison operations). We consider both recursive functions $\mathrm{rec}\, f(x) = e$ and polymorphic type abstractions $\Lambda\, e$ to be values. We write $e\ \_$ for type level application (e is a polymorphic expression). We use fold and unfold to fold and unfold elements of recursive types. Memory locations ℓ are values of reference types. The expression $!\, e$ reads the memory location e evaluates to, and $e \leftarrow e'$ is an assignment of the value computed by e' to the memory location computed by e. The expression $\mathrm{fork}\ \{e\}$ is for forking off a new thread to compute e and we write $\mathrm{cas}(e, e', e'')$ for the compare-and-set operation. A continuation, $\mathrm{cont}(K)$, is essentially a suspended evaluation context (see the operational semantics below).

Evaluation contexts of $\mathsf{F}^{\mu, ref}_{conc, cc}$ are as follows:

$$\begin{aligned} K ::= {} & - \mid K\ e \mid v\ K \mid K\ \_ \mid \mathrm{fold}\, K \mid \mathrm{unfold}\, K \mid \mathrm{if}\, K\, \mathrm{then}\, e\, \mathrm{else}\, e \mid (K, e) \mid (v, K) \mid \pi_i\, K \mid \mathrm{inj}_i\, K \\ & \mid \mathrm{match}\, K\, \mathrm{with}\, \mathrm{inj}_i\, x \Rightarrow e_i\, \mathrm{end} \mid \mathrm{ref}(K) \mid\, !\, K \mid K \leftarrow e \mid v \leftarrow K \mid \mathrm{cas}(K, e, e) \\ & \mid \mathrm{cas}(v, K, e) \mid \mathrm{cas}(v, v, K) \mid \mathrm{throw}\, K\, \mathrm{to}\, e \mid \mathrm{throw}\, v\, \mathrm{to}\, K \end{aligned}$$

The evaluation context $-$ is the empty evaluation context.

2.1 Typing

An excerpt of the typing rules is depicted in Figure 1. The context $\Xi = \alpha_1, \ldots, \alpha_n$ is a list of distinct type variables and the context $\Gamma = x_1 : \tau_1, \ldots x_n; \tau_n$ assigns types to program variables.

$$((\mathsf{rec}\, f(x) = e)\, v, \sigma) \to_K (e[v, (\mathsf{rec}\, f(x) = e)/x, f], \sigma) \qquad (\mathsf{unfold}\,(\mathsf{fold}\, v), \sigma) \to_K (v, \sigma)$$

$$((\Lambda\, e)\ \_, \sigma) \to_K (e, \sigma) \qquad (\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3, \sigma) \to_K (e_2, \sigma) \qquad (\pi_1\,(v_1, v_2), \sigma) \to_K (v_1, \sigma)$$

$$\frac{\ell \notin \mathrm{dom}(\sigma)}{(\mathsf{ref}(v), \sigma) \to_K (\ell, \sigma \uplus \{\ell \mapsto v\})} \qquad \frac{\sigma = \sigma' \uplus \{\ell \mapsto v'\}}{(\ell \leftarrow v, \sigma) \to_K ((), \sigma' \uplus \{\ell \mapsto v\})} \qquad \frac{v = \sigma(\ell)}{(!\,\ell, \sigma) \to_K (v, \sigma)}$$

$$\frac{\sigma = \sigma' \uplus \{\ell \mapsto v\}}{(\mathsf{cas}(\ell, v, v'), \sigma) \to_K (\mathsf{true}, \sigma' \uplus \{\ell \mapsto v'\})} \qquad \frac{\sigma = \sigma' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\mathsf{cas}(\ell, v, v'), \sigma) \to_K (\mathsf{false}, \sigma)}$$

$$(\mathsf{call/cc}\,(x.\, e), \sigma) \to_K (e[\mathsf{cont}(K)/x], \sigma)$$

Fig. 2. An excerpt of the head-reduction rules

The notation $K : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi \mid \Gamma; \tau')$ means that the evaluation context K satisfies the property that $\Xi \mid \Gamma \vdash K[e] : \tau'$ holds whenever $\Xi \mid \Gamma \vdash e : \tau$ does.

2.2 Operational semantics

We define the call-by-value small-step operational semantics of $\mathsf{F}^{\mu,ref}_{conc,cc}$ in two stages. We first define a head-step relation $\to_K$. Here, K is the context under which the head step is being performed. Based on this, we define the operational semantics of programs by what we call *the thread-pool step relation* $\to$. A thread pool reduces by making a head reduction step in one of the threads, by forking off a new thread, or by resuming a captured continuation:

$$\frac{(e, \sigma) \to_K (e', \sigma')}{(\vec{e}_1, K[e], \vec{e}_2; \sigma) \to (\vec{e}_1, K[e'], \vec{e}_2; \sigma')} \qquad (\vec{e}_1, K[\mathsf{fork}\ \{e\}\ ], \vec{e}_2; \sigma) \to (\vec{e}_1, K[()], \vec{e}_2, e; \sigma)$$

$$(\vec{e}_1, K[\mathsf{throw}\, v\, \mathsf{to}\, \mathsf{cont}(K')], \vec{e}_2; \sigma) \to (\vec{e}_1, K'[v], \vec{e}_2; \sigma)$$

Here, σ is the physical state of the program, *i.e.*, the program heap, which is a finite partial map from memory locations to values. An excerpt of the head-step relation is given in Figure 2. Notice that the head-step for call/cc captures the continuation that is the index of the head-step relation.

Contextual refinement/equivalence. A program e contextually refines a program e' of type τ if both programs have type τ and no *well-typed* context (a closed top-level program with a hole) can distinguish a situation where e' is replaced by e.

$$\Xi \mid \Gamma \vdash e \le_{\mathrm{ctx}} e' : \tau \triangleq \Xi \mid \Gamma \vdash e : \tau \wedge \Xi \mid \Gamma \vdash e' : \tau \wedge$$
$$\forall C.\ C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\cdot \mid \cdot; 1) \wedge C[e] \Downarrow \Rightarrow C[e'] \Downarrow$$

where

$$e \Downarrow \triangleq \exists v, \vec{e}, \sigma.\ (e; \emptyset) \to^* (v, \vec{e}; \sigma)$$

The intuitive explanation above for contextual refinement is the reason why in a contextual refinement $e \le_{\mathrm{ctx}} e'$ or in a logical relatedness relation $e \le_{\log} e'$, usually, the program on the left hand side, e, is referred to as the implementation side and the program on the right hand side, e', is referred to as the specification side.

Two programs are contextually equivalent, if each contextually refines the other:

$$\Xi \mid \Gamma \vdash e \approx_{\mathrm{ctx}} e' : \tau \triangleq \Xi \mid \Gamma \vdash e \le_{\mathrm{ctx}} e' : \tau \wedge \Xi \mid \Gamma \vdash e' \le_{\mathrm{ctx}} e : \tau$$

3 LOGICAL RELATIONS

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]. The logic of Iris is rich enough to allow for a direct inductive specification of the logical relations for programming languages with advanced features such as higher-order references, recursive types, and concurrency [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Timany et al. 2018].

3.1 An Iris primer

Iris [Jung et al. 2016, 2015; Krebbers et al. 2017a] is a state-of-the-art higher-order concurrent separation logic designed for verification of programs.

In Iris one can quantify over the Iris types κ:

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

Here *Ectx*, *Var*, *Expr* and *Val* are Iris types for evaluation contexts, variables, expressions and values of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},cc}$. Natural numbers, $\mathbb{N}$, and Booleans $\mathbb{B}$ are also included among the base types of Iris. Iris also features partial maps with finite support $\kappa \xrightarrow{\text{fin}} \kappa$ and finite sets, $\mathsf{finset}(\kappa)$. Resources in Iris are represented using partial commutative monoids, *Monoid*, and instances of resources are named using so-called ghost-names, *Names*. Finally, and most importantly, there is a type of Iris propositions *iProp*. The grammar for Iris propositions is as follows:

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\ \Phi(x) \mid \exists x : \kappa.\ \Phi(x) \\ \mid \triangleright P \mid \mu r.P \mid \Box P \mid \mathsf{wp}\ e\ \{x.\ P\} \mid \{P\}\ e\ \{x.\ Q\} \mid \Rrightarrow P \mid \boxed{P}^{\mathcal{N}} \mid \dots$$

Here, $\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$ are the standard higher-order logic connectives. The predicates Φ are Iris predicates, *i.e.*, terms of type $\kappa \to \mathit{iProp}$.

The connective $*$ is the separating conjunction. Intuitively, $P * Q$ holds if resources can be split into two *disjoint* pieces such that one satisfies P and the other Q. The magic wand connective $P \mathrel{-\!\!*} Q$ is satisfied by resources such that when these resources are combined with some resource satisfying P the resulting resources would satisfy Q.

The $\triangleright$ modality, pronounced "later" is a modality that intuitively corresponds to some abstract form of step-indexing [Appel and McAllester 2001; Appel et al. 2007; Dreyer et al. 2011]. Intuitively, $\triangleright P$ holds if P holds one step into the future. Iris has support for taking fixed points of *guarded* propositions, $\mu r.P$. This fixed point can only be defined if all occurrences of r in P are guarded, *i.e.*, appear under a $\triangleright$ modality. We use guarded fixed points for defining the interpretation of recursive types in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},cc}$. For any proposition P we have $P \vdash \triangleright P$.

When the modality $\Box$ is applied to a proposition P, the non-duplicable resources in P are forgotten, and thus $\Box P$ is "persistent." In general, we say that a proposition P is *persistent* if $P \dashv\vdash \Box P$ (where $\dashv\vdash$ is the logical equivalence of Iris propositions). A key property of persistent propositions is that they are duplicable: $P \dashv\vdash P * P$. The type system of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},cc}$ is not a sub-structural type system and variables (in the typing environment) may be used multiple times. Therefore when we interpret types as logical relations in Iris, those relations should be duplicable. We use the persistence modality $\Box$ to ensure this.

Iris facilitates specification and verification of programs by means of weakest-preconditions $\mathsf{wp}\ e\ \{v.\ P\}$, which intuitively hold whenever e is *safe* and, moreover, whenever e terminates with a resulting value v, then $P[v/x]$ holds. When x does not appear in P we write $\mathsf{wp}\ e\ \{x.\ P\}$ as $\mathsf{wp}\ e\ \{P\}$. Also, we sometimes write $\mathsf{wp}\ e\ \{\Phi\}$ for $\mathsf{wp}\ e\ \{x.\ \Phi(x)\}$

In Iris, Hoare triples are defined in terms of weakest preconditions like this: $\{P\}\, e\, \{v.\ Q\} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{wp}\, e\, \{v.\ Q\}\right)$. Note that the $\Box$ modality ensures that the Hoare triples are persistent and hence duplicable (in separation logic jargon, Hoare triples should just express "knowledge" and not claim ownership of any resources).

A key feature of Iris (as for other concurrency logics) is that specification and verification is done *thread-locally*: the weakest precondition only describes properties of execution of a single thread. Concurrent interactions are abstracted and reasoned about in terms of resources (rather than by explicit reasoning about interleavings). For programming languages that do not include continuations or other forms of non-local control flow, the weakest precondition is not only thread-local, but also what we may call *context-local*. Context-local means that to reason about an expression in an evaluation context, it suffices to reason about the expression in isolation, and then separately about what the context does to the resulting value. This form of context-locality is formally expressed by the soundness of the following bind rule

$$\textsc{inadmissible-bind}\quad \frac{\mathsf{wp}\, e\, \{v.\ \mathsf{wp}\, K[v]\, \{\Phi\}\}}{\mathsf{wp}\, K[e]\, \{\Phi\}}$$

This rule is not sound when expressions include call/cc since call/cc captures the evaluation context and hence its behaviour depends on it. See the accompanying technical appendix for a proof of inadmissibility of this rule.

Thus, for reasoning about $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ we cannot use the "standard" Iris rules [Jung et al. 2016, 2015; Krebbers et al. 2017a,b] for weakest preconditions. Instead, we use new rules such as the following:

$$\textsc{fst-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K[v]\, \{\Phi\}}{\mathsf{wp}\, K[\pi_1\,(v,w)]\, \{\Phi\}} \qquad \textsc{if-true-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K[e]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{if\ true\ then}\ e\ \mathsf{else}\ e']\, \{\Phi\}} \qquad \textsc{rec-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K[e[\mathsf{rec}\, f(x) = e, v/f, x]]\, \{\Phi\}}{\mathsf{wp}\, K[(\mathsf{rec}\, f(x) = e)\, v]\, \{\Phi\}}$$

$$\textsc{callcc-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K[e[\mathsf{cont}(K)/x]]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{call/cc}\,(x.\ e)]\, \{\Phi\}} \qquad \textsc{throw-wp}\quad \frac{\triangleright\, \mathsf{wp}\, K'[v]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{throw}\, v\, \mathsf{to\ cont}(K')]\, \{\Phi\}}$$

The difference from the standard rules is that our new rules include an explicit context K. Earlier, such rules could be derived using the bind rule, but that is not sound in general in our settings. Note that the context is used in the rules CALLCC-WP and THROW-WP for call/cc and throw. These two rules directly reflect the operational semantics of call/cc and throw.

In summa, for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ we use new non-context-local rules for reasoning about weakest preconditions, and the non-context-local rules allow us to reason about call/cc *and* throw.

Because of the explicit context K, the non-context-local rules for weakest preconditions are somewhat more elaborate to use than the corresponding context-local rules. However, that is the price we have to pay to be able to reason in general about non-local control flow. In Section 4 we will see how we can still recover a form of context-local weakest precondition for reasoning about those parts of the program that do not use non-local control flow.

In the rules above, the antecedent is only required to hold a step of computation later ($\triangleright$) — that is because these rules correspond to expressions performing a reduction step.

The update modality $\Rrightarrow$ accounts for updating (allocation, deallocation and mutation) of resources.[1] Intuitively, $\Rrightarrow P$ is satisfied by resources that can be updated to new resources for which P holds. For any proposition P, we have that $P \vdash \Rrightarrow P$. If P holds, then resources can be updated (trivially) so as to have that P holds. The update modality is idempotent, $\Rrightarrow\Rrightarrow P \dashv\vdash \Rrightarrow P$. We write

[1] This modality is called the fancy update modality in Krebbers et al. [2017a].

$P \Rrightarrow Q$ as shorthand for $P \twoheadrightarrow \Rightarrow Q$. Crucially, resources can be updated throughout a proof of weakest preconditions:

$$\Rightarrow \mathsf{wp}\, e\, \{\Phi\} \dashv\vdash \mathsf{wp}\, e\, \{\Phi\} \dashv\vdash \mathsf{wp}\, e\, \{v.\ \Rightarrow \Phi(v)\}$$

Iris features invariants $\boxed{P}^{\mathcal{N}}$ for enforcing concurrent protocols. Each invariant $\boxed{P}^{\mathcal{N}}$ has a name, $\mathcal{N}$, associated to it. Names are used to keep track of which invariants are open.[2] Intuitively, $\boxed{P}^{\mathcal{N}}$ states that P always holds. The following rules govern invariants.

$$\textsc{inv-alloc}\ \frac{\triangleright P}{\Rightarrow \boxed{P}^{\mathcal{N}}} \qquad \textsc{inv-open-wp}\ \frac{\boxed{R}^{\mathcal{N}} \qquad (\triangleright R) \twoheadrightarrow \mathsf{wp}\, e\, \{y.\ (\triangleright R) * \mathsf{wp}\, K[y]\, \{x.\ Q\}\}}{\mathsf{wp}\, K[e]\, \{x.\ Q\}}\ e \text{ is atomic}$$

These rules say that invariants can always be allocated by giving up the resources being protected by the invariant and they can be kept opened only during execution of *physically atomic* operations. Iris invariants are impredicative, *i.e.*, they can state P holds invariantly for any proposition P, including invariants. This is why the later operator is used as a guard to avoid self-referential paradoxes [Krebbers et al. 2017a]. Invariants essentially express the knowledge that some proposition holds invariantly. Hence, invariants are always persistent, *i.e.*, $\boxed{P}^{\mathcal{N}} \dashv\vdash \Box \boxed{P}^{\mathcal{N}}$.

3.2 Resources used in defining logical relations

We need some resources in order to define our logical relations in Iris. We need resources for representing memory locations of the implementation side, the memory locations of the specification side and the expression being evaluated on the specification side. These resources are written as follows:

- $\ell \mapsto_i v$: memory location ℓ contains value v on the implementation side.
- $\ell \mapsto_s v$: memory location ℓ contains value v on the specification side.
- $j \Rightarrow e$: the thread j on the specification side is about to execute e.

These resources are defined using more primitive resources in Iris, but we omit such details here. What is important is that we can use these resources to reason about programs. In particular, we can derive the following rules (and similarly for other basic expressions) for weakest preconditions and for execution on the specification side.

$$\frac{\forall \ell.\ \ell \mapsto_i v \twoheadrightarrow \mathsf{wp}\, K[\ell]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{ref}(v)]\, \{\Phi\}} \qquad \frac{\ell \mapsto_i v \twoheadrightarrow \mathsf{wp}\, K[v]\, \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[!\,\ell]\, \{\Phi\}}$$

$$\frac{\ell \mapsto_i w \twoheadrightarrow \mathsf{wp}\, K[()]\, \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\, K[\ell \leftarrow w]\, \{\Phi\}} \qquad \frac{j \Rightarrow K[\mathsf{ref}(v)]}{\Rightarrow \exists \ell.\ \ell \mapsto_s v * j \Rightarrow K[\ell]} \qquad \frac{\ell \mapsto_s v \qquad j \Rightarrow K[!\,\ell]}{\Rightarrow \ell \mapsto_s v * j \Rightarrow K[v]}$$

$$\frac{\ell \mapsto_s v \qquad j \Rightarrow K[\ell \leftarrow w]}{\Rightarrow \ell \mapsto_s w * j \Rightarrow K[()]}$$

These resources are all exclusive in the sense that:

$$\ell \mapsto_i v * \ell \mapsto_i v' \vdash \bot \qquad \ell \mapsto_s v * \ell \mapsto_s v' \vdash \bot \qquad j \Rightarrow e * j \Rightarrow e' \vdash \bot$$

[2] Officially in Iris, the update modality is in fact annotated with so-called masks (sets of invariant names), which are used to ensure that invariants are not re-opened. For simplicity, we do not include masks in this paper.

$$\text{Observational refinement: } O : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$$

$$O(e, e') \triangleq \forall j.\ j \Mapsto e' \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\exists w.\ j \Mapsto w\}$$

$$\text{Value interpretation of types: } [\![\Xi \vdash \tau]\!]_\Delta : \mathit{Val} \times \mathit{Val} \to \mathit{iProp} \text{ for } \Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$$

$$[\![\Xi \vdash \alpha]\!]_\Delta \triangleq \Delta(\alpha)$$

$$[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v, v') \triangleq v = v' = \mathsf{true} \lor v = v' = \mathsf{false}$$

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

$$[\![\Xi \vdash \mathsf{cont}(\tau)]\!]_\Delta(v, v') \triangleq \exists K, K'.\ v = \mathsf{cont}(K) \land v' = \mathsf{cont}(K') \land \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K')$$

$$\text{Evaluation context interpretation of types: } \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Ectx} \times \mathit{Ectx} \to \mathit{iProp} \text{ for } \Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$$

$$\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K') \triangleq \forall v, v'.\ [\![\Xi \vdash \tau]\!]_\Delta(v, v') \Rightarrow O(K[v], K'[v'])$$

$$\text{Expression interpretation of types: } \mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp} \text{ for } \Delta : \mathit{Var} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$$

$$\mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(e, e') \triangleq \forall K, K'.\ \mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta(K, K') \Rightarrow O(K[e], K'[e'])$$

$$\text{Logical relatedness: } \Xi \mid \Gamma \vDash e \le_{\log} e' : \tau : \mathit{iProp}$$

$$\Xi \mid \Gamma \vDash e \le_{\log} e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v'}.\ \left(\mathop{\Large *}_{x_i:\tau_i} [\![\Xi \vdash \tau_i]\!]_\Delta(v_i, v_i')\right) \Rightarrow \mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$
$$\text{if } \Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$$

Fig. 3. An excerpt of the logical relations for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$

3.3 Logical relations in Iris

Figure 3 presents our binary logical relation for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$. We define the logical relation in several stages. The first thing we define is the relation of observational refinement. Intuitively, an expression e observationally refines an expression e' if, whenever e reduces to a value so does e'. We define this in Iris using magic wand and weakest precondition. The whole formula reads as follows: Assuming that there is some thread j on the specification side that is about to execute e' (represented in Iris by $j \Mapsto e'$) then, after execution of e, we know that thread j on the specification side has also been executed to some value w.

We then use the notion of observational refinement defined above to define the value relation, the expression relation and the evaluation context relation for each type. In contrast to earlier definitions of logical relations in Iris [Krebbers et al. 2017b; Krogh-Jespersen et al. 2017; Timany et al. 2018], our logical relation is an example of so-called biorthogonal logical relations [Pitts 2005], also known as top-top closed logical relations. That is, we define two expressions to be related if plugging them into related evaluation contexts results in observationally related expressions. Two evaluation contexts are defined to be related if plugging related values into them results in observationally related expressions.

The value relation interpretation $[\![\Xi \vdash \tau]\!]_\Delta$ of a type in context is defined by induction on τ. Here Δ is an environment mapping type variables in Ξ to Iris relations. For all the non-continuation types, the definition is exactly as in for the language without call/cc, see Krebbers et al. [2017b], and thus we only include the cases for type variables, Booleans and function types in addition to the new case for the continuation types (the full definition can be found in the accompanying technical appendix).

Two values of the Boolean type $\mathbb{B}$ are related if they are both true or they are both false. The value relation for functions types $\tau \to \tau'$ expresses that two values of the function type are related if whenever applied to related values of the domain type, τ, the resulting *expressions* are related at the codomain type, τ'. The use of the *persistently modality*, $\square$, here is to make sure that the interpretations are persistent. Finally, the relational interpretation of $\mathrm{cont}(\tau)$ expresses that two continuations are related whenever their corresponding evaluation contexts are related at the evaluation context relation for the type in question.

The evaluation context relation $\mathcal{K}[\![\Xi \vdash \tau]\!]_\Delta$ relates evaluation contexts K and K' if plugging related values of type τ in them results in observationally related expressions.

The expression relation is the standard biorthogonal expression relation. It states that $\mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(e, e')$ holds whenever, for any two related evaluation contexts K and K', the expressions $K[e]$ and $K'[e']$ are observationally related.

The notion of logical relatedness states, as usual for call-by-value languages, that two expressions e and e' are logically related if substituting related values for their free variables results in related expressions.

We can now state and prove the fundamental theorem of logical relations for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$. The theorem expresses that any well-typed expression is logically related to itself.

Theorem 3.1 (Fundamental theorem of logical relations).

$$\Xi \mid \Gamma \vdash e : \tau \Rightarrow \Xi \mid \Gamma \vDash e \le_{\mathit{log}} e : \tau$$

This theorem is proven by induction on the typing derivation using the basic rules for weakest-preconditions and executions on the specification side.

The above theorem, together with some basic properties of observational refinement, implies the soundness of our logical relations, *i.e.*, that logical relatedness implies contextual refinement:

Theorem 3.2 (Soundness of logical relations).

$$\Xi \mid \Gamma \vDash e \le_{\mathit{log}} e' : \tau \Rightarrow \Xi \mid \Gamma \vDash e \le_{\mathit{ctx}} e' : \tau$$

Our logical relation is expressed in terms of weakest preconditions and the proofs of the above theorems use the earlier presented proof rules for weakest preconditions. Before turning to applications, we pause to present *context-local weakest preconditions*, which we can use to simplify reasoning about program fragments, which do not use non-local control flow.

4 CONTEXT-LOCAL WEAKEST PRECONDITIONS (CLWP)

To make it simpler to reason about expressions that do not use non-local control flow, we define a new notion of *context-local weakest precondition*. The definition is given in terms of the earlier weakest precondition, which, as we will explain below, means that we will be able to mix and match reasoning steps using (non-context local) weakest preconditions and context-local weakest preconditions.

Definition 4.1. The *context-local weakest precondition* of e wrt. Φ is defined as:

$$\mathsf{clwp}\ e\ \{\Phi\} \triangleq \forall K, \Psi.\ (\forall v.\ \Phi(v) \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ \{\Psi\}) \mathrel{-\!\!*} \mathsf{wp}\ K[e]\ \{\Psi\}$$

Note how the above definition essentially says that $\mathsf{clwp}\ e\ \{\Phi\}$ holds if the bind rule holds for e, which intuitively means that e does not use non-local control flow. Therefore, the bind rule is sound for context-local weakest preconditions:

$$\textsc{bind}\quad \frac{\mathsf{clwp}\ e\ \{v.\ \mathsf{clwp}\ K[v]\ \{\Phi\}\}}{\mathsf{clwp}\ K[e]\ \{\Phi\}}$$

Moreover, the "standard" rules for the basic language constructs (excluding call/cc and throw, of course) can also be derived for context-local weakest preconditions: Here is an excerpt of the rules that we can derive:

$$\textsc{fst-clwp}\quad \frac{\triangleright\ \mathsf{clwp}\ v\ \{\Phi\}}{\mathsf{clwp}\ \pi_1\ (v, w)\ \{\Phi\}} \qquad \textsc{if-true-clwp}\quad \frac{\triangleright\ \mathsf{clwp}\ e\ \{\Phi\}}{\mathsf{clwp}\ \mathsf{if\ true\ then}\ e\ \mathsf{else}\ e'\ \{\Phi\}} \qquad \textsc{rec-clwp}\quad \frac{\triangleright\ \mathsf{clwp}\ e[\mathsf{rec}\ f(x) = e, v/f, x]\ \{\Phi\}}{\mathsf{clwp}\ (\mathsf{rec}\ f(x) = e)\ v\ \{\Phi\}}$$

$$\textsc{alloc-clwp}\quad \mathsf{clwp}\ \mathsf{ref}(v)\ \{w.\ \exists \ell. w = \ell * \ell \mapsto_i v\} \qquad \textsc{load-clwp}\quad \frac{\triangleright\ \ell \mapsto_i v}{\mathsf{clwp}\ !\ell\ \{w.\ w = v * \ell \mapsto_i v\}}$$

We can also use invariants during atomic steps of computation while proving context-local weakest preconditions:

$$\textsc{inv-open-clwp}\quad \frac{\boxed{R}^{\mathcal{N}} \qquad (\triangleright R) \mathrel{-\!\!*} \mathsf{clwp}\ e\ \{v.\ (\triangleright R) * Q\} \qquad e\ \text{ is atomic}}{\mathsf{clwp}\ e\ \{v.\ Q\}}$$

Now we have both (non-context-local) weakest preconditions and context-local weakest preconditions. What is the upshot of this? The key point is that when we prove correctness / relatedness of programs, we can use the simpler context-local weakest preconditions for reasoning about those parts of the program which are context local (do not use call/cc or throw) and only use the (non-context-local) weakest preconditions for reasoning about those parts that may involve non-local control flow. This fact is expressed formally by the following derivable rule, which establishes a connection between weakest-preconditions and context-local weakest preconditions.

$$\textsc{clwp-wp}\quad \frac{\mathsf{clwp}\ e\ \{\Psi\} \qquad \forall v.\ \Psi(v) \mathrel{-\!\!*} \mathsf{wp}\ K[v]\ \{\Phi\}}{\mathsf{wp}\ K[e]\ \{\Phi\}}$$

This rule basically says that if we know that e *context-locally* guarantees postcondition Ψ then we can prove $\mathsf{wp}\ K[e]\ \{\Phi\}$ by assuming that *locally*, under the context K, it will only evaluate to values that satisfy Ψ. Moreover, it guarantees that evaluation of e does not tamper with the evaluation context that we are considering it under.

Similarly to Hoare-triples above, we define context-local Hoare-triples based on context-local weakest preconditions:

$$\{P\}^{\mathsf{cl}}\ e\ \{v.\ Q\} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{clwp}\ e\ \{v.\ Q\}\right)$$

5 WEB SERVER REFINEMENT

It is well-known that web servers are intricate to program, in particular because they have to keep track of the complex evolution of the state of clients. Clients can refresh pages, press back and forward buttons of the browser, and so forth. Both researchers [Flatt 2017; Krishnamurthi et al. 2007; Queinnec 2004] and practitioners [Hendershott 2017; Might 2017] have therefore advocated that one can simplify web server implementations considerably by using explicitly captured (using

```
let handler1 :  ServerConnT -> 1 =
  let tb = newTable () in
  fun (cn : ServerConnT) ->
    let (reader, writer) = cn in
    match reader () with
      (Some cid, n) ->
      begin
        match get tb cid with
          None -> ()
            (* unknown resumption id! *)
        | Some sum ->
          writer (result (sum + n));
          writer
            (resumptionid
              (associate tb (sum + n)));
          abort
      end
    | (None, n) ->
      writer (result n);
      let cid = associate tb n
      in writer (resumptionid cid)
```

```
let read_client tb writer =
  callcc (k. writer
    (resumptionid (associate tb k)); abort)

let handler2 : ServerConnT -> 1 =
  let tb = newTable () in
  fun (cn : ServerConnT) ->
    let (reader, writer) = cn in
    match reader () with
      (Some cid, n) ->
      begin
        match get tb cid with
        | None -> ()
            (* unknown resumption id! *)
        | Some k ->
            throw (n, reader, writer) to k
      end
    | (None, n) ->
      let rec loop m reader writer =
        writer (result m);
        let (v, reader, writer) =
          read_client tb writer
        in
        loop (m + v) reader writer
      in loop n reader writer
```

Fig. 4. Two server handlers: one storing the state of the server explicitly (left) and one storing the continuation (right)

call/cc) server-side continuations. The point is that using continuations simplifies the book-keeping of the clients' state and hence allows for a more direct style implementation of web servers, where the interaction with clients can be programmed as though one was communicating through a console.

This *continuation-based* approach to web server implementation is in contrast to the perhaps more common practice, which we refer to as *state-storing*, where for every request, the server needs to analyze its internally stored state along with the client request in order to determine the proper response. In the *continuation-based* approach, the server simply resumes its internally stored continuation when it gets a new request from the client. The Racket web development community [Flatt 2017] is probably the most prominent user of continuation-based servers.

5.1 Two Servers

Figure 4 shows implementations of two handlers mimicking rudimentary web servers. We use an ML-like syntax for the sake of brevity and legibility, but all our example programs can be written in the syntax of our programming language, $\mathsf{F}^{\mu,ref}_{conc,cc}$, and that is indeed what we have done in our Coq formalization. Given a connection, `serverConnT`, a pair of functions for reading and writing, each handler will sum up the numbers given by the client so far, and return the sum back to the client together with a *resumption id*. The client may chose to resume an old computation by giving a new number along with a resumption id or simply make a request to start a computation by giving the first number in the sum to be computed.

The program `handler1` is a store-based implementation, which stores each state (the sum so far) together with a resumption id in a table. The program `handler2` is a continuation-based implementation. It simply implements a loop in a fashion as though the user interaction is taking place over a terminal rather than between a client and a server. This loop prints the sum so far and subsequently reads from the client. The operation of reading a value from the client is implemented using the call/cc command to capture the current continuation. The captured continuation is associated to a resumption id which is given to the client, so that it may continue the computation by providing a new value to be added to the current sum along with the resumption id in question. Note that since after "reading from the client", we will be communicating with the client on a different connection, we need the `read_client` function to return the new connection as well as the "read value".[3] See Queinnec [2004] and Krishnamurthi et al. [2007] for more details on how these kinds of web servers are implemented and used.

One important advantage of using the continuation-based server implementation strategy is scalability. In our rudimentary example in Figure 4, the state of the server for each client is primitively simple: the current sum! In general, the state of the web server can be fairly complex. Indeed, the complexity of the state is in some cases the main reason for functionality flaws in web applications [Krishnamurthi et al. 2007]. Independent of the intricacy of the state, the server implementation for serving returning clients remains the same: resuming the captured continuation.

A server program using either of our two handlers could be implemented as follows:

```
let rec serve (listener : 1 -> ServerConnT) (handler : ServerConnT -> 1) : 1 =
  let v = listener () in fork {handler v};  serve listener handler
```

This server program accepts a listener (a function returning a connection) and a handler. It loops, waiting for connections. For each connection it creates a new thread and hands the connection over to its handler. This server program can be applied to any proper listener and either of the handlers depicted in Figure 4.

The following is an example of a client program that can interface with the above server (when instantiated with either handler):

```
1  let send_receive cid number =
2    let (reader1, writer1, reader2, writer2) = newConnection () in
3    contact_server (reader2, writer1); writer1 (cid, number); reader2 ()
4
5  let client () =
6    let (cid1, ans1) = send_receive None 10 in (* ans1 = 10 *)
7    let (cid2, ans2) = send_receive (Some cid1) 5 in (* ans2 = 15 *)
8    let (cid3, ans3) = send_receive (None) 17 in (* ans3 = 17 *)
9    let (cid4, ans4) = send_receive (Some cid3) 9 in (* ans4 = 26 *)
10   let (cid5, ans5) = send_receive (Some cid1) 19 in (* ans5 = 29 *)
11   let (cid6, ans6) = send_receive (Some cid2) 17 in (* ans6 = 32 *)
12   ()
```

It shows that a client can indeed go back and forth on the state, *e.g.*, by pressing the back button in the browser. For instance, the resumption id `cid1` is resumed twice, once on line 7 and once on line 10, with a few interactions there in between. In this program the function `send_receive` creates a new connection, sends it to the server (establishing a connection to the server) and subsequently makes the request and retrieves the response from the server and returns it.

In this section we show that the two server implementations, the continuation-based implementation and the state-storing implementation, are contextually equivalent. Note that our server implementations are parameterized on a pair of functions, one for reading requests from the client

[3]The command **abort** is the command that ends the program (thread) and can be written in our programming language as **throw** () **to** - where - is the empty evaluation context.

and one for writing to the client. The idea is that these functions are an abstraction of a TCP connection and thus the contextual equivalence can be understood as showing that clients cannot distinguish between the two implementations.

The crux of the proof of contextual equivalence is proving that the two handlers in Figure 4 are contextually equivalent. Both of these handlers start out by establishing an empty table for storing their resumptions. In the state-storing implementation, the table is used to store the state (the sum so far), while in the continuation-based implementation, the table stores the continuation. After creating the tables, both implementations return functions which are the actual handlers. These functions internally use their respective tables to store and look-up resumptions. The table implementation itself is straightforward and thus omitted. It uses a spin lock (omitted) for synchronization. Since the table and the lock do not make use of call/cc and throw, we employ context-local weakest preconditions to give relational specifications for them. Hence we can reason about the table and lock implementations in the way we usually do in Iris for concurrent programs without continuations. When proving relatedness of the handlers, which we do using (non context-local) weakest preconditions, the CLWP-WP rule allows us to make use of the context-local relational specifications of the table and lock.

In the rest of this section we will first present and discuss our relational specifications for the table and the lock. After that we will discuss the logical relatedness of the two handlers. Our relational specifications for the table and the lock are stronger than the specifications one usually encounters in the literature. We need this strengthening because the continuations stored in the table refers to the table itself in the continuation-based implementation. This is, fundamentally, also the reason why, although the table code is identical in both handlers, we cannot use the fundamental theorem of logical relations to conclude that they are related in a sufficiently strong way.

5.2 Relational spec for the table and the lock

We now discuss the relational specifications for the tables and the locks that they use for synchronization. All the reasoning in this subsection is context-local, using the primitive rules for context-local weakest preconditions. The table specifications are used in the proof of relatedness of the handlers, which we discuss in the following subsection.

The essence of relating the tables on both sides (specification side and implementation side) is simple. The contents of new tables are related (as they are both empty) and we only store values that are suitably related. Hence, when looking the table up we are guaranteed to receive related values, if any. This is formally captured in the following relational specifications:

$$\{j \Rrightarrow K[\mathbf{newTable}\ ()]\}^{\mathrm{cl}}$$
$$\quad \mathbf{newTable}\ ()$$
$$\{v.\ \exists v'.\ j \Rrightarrow K[v'] * \forall \Phi.\ \Rrightarrow \exists \gamma.\ relTables(v, v', \gamma, \Phi)\}$$

$$\{relTables(tb, tb', \gamma, \Phi) * j \Rrightarrow K[\mathbf{get}\ tb'\ n]\}^{\mathrm{cl}}$$
$$\quad \mathbf{get}\ tb\ n$$
$$\{v.\ \exists v'.\ j \Rrightarrow K[v'] * \left(v = v' = None \lor (\exists w, w'\ v = Some(w) \land v' = Some(w') * \Phi(v, v'))\right)\}$$

$$\{relTables(tb, tb', \gamma, \Phi) * \Phi(v, v') * j \Rrightarrow K[\mathbf{associate}\ tb'\ v]\}^{\mathrm{cl}}$$
$$\quad \mathbf{associate}\ tb\ v$$
$$\{v.\ \exists n.\ v = n * j \Rrightarrow K[n]\}$$

The specifications for **get** and **associate** are exactly as we explained above. The most important part of this spec is the persistent proposition $\mathit{relTables}(v, v', \gamma, \Phi)$ which intuitively says that the tables v and v' have contents that are pair-wise related by the binary predicate Φ. The name γ for ghost resources is used for synchronization purposes. The specification of **newTable** is *stronger* than usual in that it guarantees that for any user picked predicate we can obtain that the two tables are related. Contrast this with the weaker standard style specification

$$\begin{array}{l}\forall \Phi.\ \{j \Rrightarrow K[\textbf{newTable}\ ()]\}^{\mathrm{cl}} \qquad \text{(weaker standard spec)}\\ \quad \textbf{newTable}\ ()\\ \quad \{v.\ \exists v'.\ j \Rrightarrow K[v'] * \exists \gamma.\ \mathit{relTables}(v, v', \gamma, \Phi)\}\end{array}$$

which quantifies over Φ outside the whole triple.

Notice that with our stronger specification we can refer to the tables themselves in the predicate Φ that we pick for relating the contents, whereas in the (weaker standard spec) specification one has to pick this relation beforehand, and hence one cannot refer to the tables v and v' because they have not been created yet!

The predicate $\mathit{relTables}(tb, tb', \gamma, \Phi)$ is defined in terms of the *relLocks* predicate, which pertains to the relational specification of spin locks given below.

$$\begin{array}{l}\mathit{relTables}(tb, tb', \gamma, \Phi) \triangleq \mathit{relLocks}(tb.\mathit{lock}, tb'.\mathit{lock}, \gamma, P_\Phi)\\ \qquad P_\Phi \triangleq \exists ls.\ \mathit{contents}(tb, \mathit{map}\ \pi_1\ ls) * \mathit{contents}(tb', \mathit{map}\ \pi_2\ ls) * \mathop{\Large *}_{(x,x')\in ls} \Phi(x, x')\end{array}$$

Here $tb.\mathit{lock}$ is the lock associated with the table tb. The proposition P_Φ above simply states that the there is a list of pairs of values, which are pairwise related by Φ and, moreover, that the first projections of these pairs are stored in the implementation side table and the second projections of these pairs are stored in the specification side table. The *contents* predicate simply specifies that the index of an element in the table is its index in the list.

Relational spec for the spin lock. We use the following relational specification for relating the locks used on the implementation and the specification side.

$$\{j \Rrightarrow K[\textbf{newlock}\ ()]\}^{\mathrm{cl}}\ \textbf{newlock}\ () \qquad \left\{\begin{array}{l} v.\ \exists v'.\ j \Rrightarrow K[v'] *\\ \quad \forall P.P \Rrightarrow \exists \gamma.\ \mathit{relLocks}(v, v', \gamma, P)\end{array}\right\}$$

$$\{\mathit{relLocks}(v, v', \gamma, P) * j \Rrightarrow K[\textbf{acquire}\ v']\}^{\mathrm{cl}} \quad \textbf{acquire}\ v \quad \{\_.\ j \Rrightarrow K[()] * \mathit{locked}(\gamma) * P\}$$

$$\{\mathit{relLocks}(v, v', \gamma, P) * P * \mathit{locked}(\gamma) * j \Rrightarrow K[\textbf{release}\ v']\}^{\mathrm{cl}} \quad \textbf{release}\ v \quad \{\_.\ j \Rrightarrow K[()]\}$$

The specification captures that whenever we acquire the lock on the implementation side, the lock on the specification side is free and can be acquired. This is necessary for showing contextual refinements because if the implementation side converges, then we need to show that so does the specification side and the acquire operation is potentially non-terminating. This also means that whenever we release the lock on the implementation side, the lock on the specification side is also released.

Our relational lock specification is also a bit stronger than usual, (cf. the quantification over P in the **newlock** specification), because we use the lock specification when proving the relational specification for tables described above.

The persistent proposition $\mathit{relLocks}(v, v', \gamma, P)$ states that v is a lock protecting two things: resources P *and* the fact that v' is not acquired. The proposition $\mathit{locked}(\gamma)$ states that both of the locks associated to γ are currently acquired.

5.3 Proving equivalence of handlers

We devote the reset of this section to discussing the main result of this section: proving relatedness of handlers in Figure 4.

Theorem 5.1.

$$\Xi \mid \Gamma \vDash \mathit{handler2} \approx_{ctx} \mathit{handler1} : \mathbf{ServerConnT} \to 1$$

Our mechanized proof of the above theorem is done by showing logical relatedness in both directions and then appealing to the soundness of the logical relation (Theorem 3.2). Here we only discuss the proof of one direction:

$$\Xi \mid \Gamma \vDash \mathit{handler2} \leq_{\log} \mathit{handler1} : \mathbf{ServerConnT} \to 1$$

We use the rules for weakest preconditions and executions on the specification side explained above and make use of the relational specification given above for tables, which is justified by the clwp-wp rule. A key element of the proof is the choice of predicate for relating the contents of the two tables. We use the following predicate:

$$\Phi_{\mathit{handlers}}(w, w') = \exists \mathit{sum} \in \mathbb{N}.\ w' = \mathit{sum} \land \exists K.\ w = \mathsf{cont}\left(K\left[\begin{array}{l}\mathsf{let}\,(v, \mathit{reader}, \mathit{writer}) = - \,\mathsf{in}\\ \mathbf{loop}\ (\mathit{sum} + v)\ \mathit{reader}\ \mathit{writer}\end{array}\right]\right)$$

It essentially states that the values that are related in the two tables are: a captured continuation, on the implementation side, and a number, on the specification side. Furthermore, there is a number, *sum*, which is intuitively the sum so far. The natural number on the specification side is exactly this *sum*. The captured continuation on the implementation side is a continuation under some evaluation context K (existentially quantified). When resumed with a new value and connection, the stored continuation calls **loop** with the new connection along with *sum* plus that value. The relation Φ_{handlers} above is indeed capturing the essence of the intuitive reason why the two implementations of handlers have contextually equivalent behavior.

According to the definition of our logical relations, to show logical relatedness we need to show that given any two related contexts the two programs behave in a related way. Since at the time of picking the predicate above we do not know what contexts we will have to operate under, we have to consider that our code of interest is inside some arbitrary (hence existentially quantified) evaluation context.

Note that the captured continuation mentioned in Φ_{handlers} refers to **loop**, which internally (see the code in Figure 4), uses the table itself. This is the reason why we need stronger relational specification for the table mentioned above.

6 ONE-SHOT CALL/CC

In this section we consider a more technical verification challenge involving continuations, due to Friedman and Haynes [1985]. The challenge is to show that call/cc can be implemented using references and one-shot continuations, *i.e.*, continuations that can only be called once. This problem has been studied for *sequential* higher-order languages with references in Dreyer et al. [2012];

Støvring and Lassen [2007], with pen-and-paper proofs, not mechanized formal verication. Here we show that the equivalence also holds in our concurrent language (subtly so; because we are using *may* contextual equivalence) and we give a mechanized formal proof thereof. Our proof is inspired by the proof of Dreyer et al. [2012], but we use a more involved invariant because of concurrency.

First, we define a polymorphic higher-order function that given a function f calls f with the current continuation:

$$\mathfrak{CC} \triangleq \Lambda\, \lambda f.\ \mathsf{call/cc}\,(x.\ f\ x)$$

Note that $\mathfrak{CC}$ has type $\cdot \mid \cdot \vdash \mathfrak{CC} : \forall\alpha.\,(\mathsf{cont}(\alpha) \to \alpha) \to \alpha$. Next, we will define a variant $\mathfrak{CC}'$ using one-shot continuations, and then prove the contextual equivalence of $\mathfrak{CC}$ and $\mathfrak{CC}'$.

To this end, we first define one-shot continuations $\mathfrak{CC}_1$ as follows:

$$\begin{aligned}\mathfrak{CC}_1 \triangleq \Lambda\, \lambda f.\ & \mathsf{let}\ b = \mathsf{false}\ \mathsf{in}\\ & \mathsf{call/cc}\,(x.\ f\ (\mathsf{cont}(\mathsf{let}\ y = -\ \mathsf{in}\ \mathsf{if}\ !b\ \mathsf{then}\ \Omega\ \mathsf{else}\ b \leftarrow \mathsf{true}; \mathsf{throw}\ y\ \mathsf{to}\ x)))\end{aligned}$$

Here Ω is the trivially diverging expression. Note that $\cdot \mid \cdot \vdash \mathfrak{CC}_1 : \forall\alpha.\,(\mathsf{cont}(\alpha) \to \alpha) \to \alpha$. When applied, the one-shot continuation, $\mathfrak{CC}_1$, first allocates a *one-shot bit* b and then calls the given function with a continuation that uses b to ensure that the continuation is only called once.

Using one-shot continuations, we now define $\mathfrak{CC}'$:

$$\begin{aligned}\mathfrak{CC}' \triangleq \Lambda\, \lambda f.\ & \mathsf{let}\ \ell = \mathsf{ref}(\mathsf{cont}(-))\ \mathsf{in}\ G\ f\\ G \triangleq\ & \mathsf{rec}\ G(f) =\\ & \quad \mathsf{let}\ x = \mathfrak{CC}_1\ \_\ (\lambda y.\ \ell \leftarrow y; f\ (\mathsf{cont}(\mathsf{throw}\ \mathsf{cont}(-)\ \mathsf{to}\ !\ell)))\ \mathsf{in}\\ & \quad \mathfrak{CC}_1\ \_\ (\lambda y.\ G\ (\lambda\_.\ \mathsf{throw}\ x\ \mathsf{to}\ y))\end{aligned}$$

The expression $\mathfrak{CC}'$ above has the same type as $\mathfrak{CC}$. $\mathfrak{CC}'$ perhaps looks fairly complex but the intuition is straightforward. It first allocates ℓ with the trivial continuation, then it takes a one-shot continuation and updates ℓ. When the one-shot continuation is used, it will first grab another *fresh* one-shot continuation and update ℓ with it before continuing. Hence, intuitively, every time the one-shot continuation stored in ℓ is used, it is immediately refreshed with an unused one, thus mimicking the behavior of $\mathfrak{CC}$.

We now prove that $\mathfrak{CC}$ is contextually equivalent to $\mathfrak{CC}'$:

Theorem 6.1.

$$\cdot \mid \cdot \vDash \mathfrak{CC} \approx_{ctx} \mathfrak{CC}' : \forall\alpha.\,(\mathsf{cont}(\alpha) \to \alpha) \to \alpha$$

We only discuss one side of the refinement, namely, $\mathfrak{CC}' \leq_{\mathrm{ctx}} \mathfrak{CC}$. The proof of the other side is similar but simpler.

Our proof is similar to the one by Dreyer et al. [2012], except for the invariant that is used to prove relatedness.[4] Translated to our setting, the invariant used by Dreyer et al. [2012] is:

$$\boxed{\exists b.\ b \mapsto_i \mathsf{false}\ *\ \ell \mapsto_i \mathsf{cont}\begin{pmatrix}\mathsf{let}\ y = -\ \mathsf{in}\ \mathsf{if}\ !\,b\ \mathsf{then}\ \Omega\ \mathsf{else}\\ b \leftarrow \mathsf{true}; \mathsf{throw}\ y\ \mathsf{to}\ \mathsf{cont}(K[\mathit{restore}(\ell)])\end{pmatrix}}^{\mathcal{N}.\mathfrak{CC}}$$

where

$$\mathit{restore}(\ell) \triangleq \mathsf{let}\ x = -\ \mathsf{in}\ \mathfrak{CC}_1\ \_\ (\lambda y.\ G\ (\lambda\_.\ \mathsf{throw}\ x\ \mathsf{to}\ y))$$

[4] In the work of Dreyer et al. [2012], invariants were called *islands*.

Here K is the continuation that is captured by $\mathfrak{CC}$. Intuitively, it states that the continuation stored in ℓ is a one-shot continuation that has never been used (as the one-shot bit b stores false). Furthermore, whenever used it will first restore ℓ with a fresh one-shot continuation (using the nested evaluation context $restore(\ell)$).

This invariant suffices for a *sequential* programming language. However, in our concurrent settting, the "continuation" captured by $\mathfrak{CC}'$ may be shared among multiple threads and, if they use it concurrently, a race may occur. In other words, it may happen that a thread is using the continuation captured by $\mathfrak{CC}'$ and before this thread manages to capture another one-shot continuation and restore ℓ, another thread attempts to use the then invalid one-shot continuation, and hence it diverges.

We prove that the contextual refinement still holds (despite the possibility of divergence). However, because of the possible racing, we need to use a weaker invariant:

$$\boxed{\begin{array}{l}\exists b, M.\ \mathit{OneShotBits}(M) * \mathit{isOneShotBit}(b) * \\ \quad \left(\ast_{r \in M} \exists v \in \{\mathsf{true}, \mathsf{false}\}\ .\ r \mapsto_i v\right) * \\ \quad \ell \mapsto_i \mathsf{cont}\left(\begin{array}{l}\mathsf{let}\ y = - \ \mathsf{in}\ \mathsf{if}\ !b\ \mathsf{then}\ \Omega\ \mathsf{else} \\ b \leftarrow \mathsf{true}; \mathsf{throw}\ y\ \mathsf{to}\ \mathsf{cont}(K[\mathit{restore}(\ell)])\end{array}\right)\end{array}}^{\mathcal{N}.\mathfrak{CC}}$$

This invariant says that ℓ stores a one-shot continuation with a one-shot bit b and that we have a set of bits that, intuitively, have been associated to one-shot continuations. We also know that b is one such one-shot bit, $\mathit{isOneShotBit}(b)$. The predicates $\mathit{OneShotBits}()$ and $\mathit{isOneShotBit}()$ are defined using iris resources. Details can be found in the accompanying technical appendix. Here we only need to know two things about them, namely that $\mathit{isOneShotBit}(b)$ is persistent and that

$$\mathit{OneShotBits}(M) * \mathit{isOneShotBit}(b) \vdash b \in M \tag{in-bits}$$

Persistence allows us to retain the information $\mathit{isOneShotBit}(b)$ once we have opened the invariant and have read ℓ. Due to the race condition explained above, when we open the invariant we know, by (in-bits), that there is a value $v \in \{\mathsf{true}, \mathsf{false}\}$ stored in b, and this suffices for being able to complete the refinement proof.

The other refinement, $\mathfrak{CC} \leq_{\mathrm{ctx}} \mathfrak{CC}'$ is simpler and basically follows using the same argument as in Dreyer et al. [2012]. This makes sense intuitively because we simply have to show that *there exists* an execution on the specification side that converges.

7 CORRECTNESS OF CONTINUATION BASED COOPERATIVE CONCURRENCY

Cooperative concurrency is a form of concurrency where threads cooperate and use a *yield* command to relinquish control to other threads. This is in contrast to *preemptive* concurrency, where the operating system preempts and schedules threads. Cooperative concurrency is often implemented using continuations. Forking a new thread suspends execution of the current thread, enqueues the suspension in a queue, and runs the forked thread. The yield command dequeues a previously enqueued suspension (thread) and resumes it, after enqueuing current continuation. Implementing cooperative concurrency using continuations is a well-known technique [Haynes et al. 1984].

Cooperative concurrency is also referred to as light-weight concurrency or green threads. These names allude to the fact that light-weight threads are more efficient to use than threads managed by the operating system. There are many libraries providing support for cooperative concurrency

in different programming languages, *e.g.*, "Goroutines" for Go , "Libmil" for C , "Mioco" for Rust , etc.[5]

In this section, we prove correctness of a continuation-based implementation of *cooperative concurrency*. It is not entirely obvious how to state the desired correctness property. Here we use a relational approach inspired by compiler correctness, and show that a language with built-in cooperative concurrency can be compiled into a continuation-based implementation of cooperative concurrency. We prove the correctness of this compilation, by showing that a compiled program is a refinement of its source program.

The programming language with built-in cooperative concurrency is called $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$. This language serves as our specification of cooperative concurrency. Concretely, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ provides two primitive commands Cfork and yield. The semantics of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ keeps track of the currently running thread and keeps executing that thread until it reaches a fork or a yield command. In the former case, it starts a new thread and starts executing that. In the latter case, the semantics picks another thread and proceeds with executing that.

The programming language that we consider as the target of translation of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ is $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$, the sequential fragment of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, *i.e.*, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ without fork and cas.

In the rest of this section, we define the precise syntax and semantics of the source and target languages, the translation of cooperative concurrency, and a cross-language logical relation between the source and target language, which we use to show the correctness of the translation.

7.1 Syntax and semantics of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$

The syntax of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ is similar to $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, except that it does not feature a cas operation and instead features the cooperative concurrency primitives Cfork and yield. The programming languages $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ have the same types and similar typing rules for all fragments of the syntax that they share. For the sake of clarity we use $\vdash_{\mathsf{coop}}$ and $\vdash_{\mathsf{seq}}$ for the typing judgement of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ respectively. The typing rules for Cfork and yield in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ are as follows:

$$\text{T-CFork}\quad \frac{\Xi \mid \Gamma \vdash_{\mathsf{coop}} e : \tau}{\Xi \mid \Gamma \vdash_{\mathsf{coop}} \mathsf{Cfork}\ \{e\} : 1} \qquad\qquad \text{T-Yield}\quad \Xi \mid \Gamma \vdash_{\mathsf{coop}} \mathsf{yield} : 1$$

The language $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ is a fragment of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ with the same operational semantics. Formally, the head steps for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ are identical to the corresponding fragment in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$. For the general execution, however, we define the *sequential step* relation, $\rightarrow_{\mathsf{seq}}$, in place of the thread-pool step $\rightarrow$ for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$.

$$\frac{(e,\sigma) \rightarrow_K (e',\sigma')}{(K[e];\sigma) \rightarrow_{\mathsf{seq}} (K[e'];\sigma')} \qquad (K[\mathsf{throw}\ v\ \mathsf{to}\ \mathsf{cont}(K')];\sigma) \rightarrow_{\mathsf{seq}} (K'[v];\sigma)$$

The head-step relation for the fragment of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ that is included in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$, *i.e.*, everything apart from Cfork and yield, is defined identically to $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$. However, for general reduction, in place of a thread-pool step we define a *cooperative step*, $\rightarrow_{\mathsf{coop}}$. The step $(\vec{e};n;\sigma) \rightarrow_{\mathsf{coop}} (\vec{e'};n';\sigma')$ is to be understood as a step transforming the thread pool $\vec{e}$ into the thread pool $\vec{e}$ and state σ into σ' while changing the current thread being executed from thread

[5] See https://golangbot.com/goroutines/, http://libmill.org and https://github.com/dpc/mioco.

number n to thread number n'. The cooperative step relation is defined below. Notice that in this definition a thread is only executed if it is the current thread.

$$\frac{(e,\sigma) \to_K (e',\sigma') \qquad \mathit{length}(e_1) = n}{(\vec{e}_1, K[e], \vec{e}_2; n; \sigma) \to_{\mathsf{coop}} (\vec{e}_1, K[e'], \vec{e}_2; n; \sigma')}$$

$$\frac{\mathit{length}(e_1) = n \qquad m = \mathit{length}(\vec{e}_1, K[\mathsf{Cfork}\ \{e\}\ ], \vec{e}_2)}{(\vec{e}_1, K[\mathsf{Cfork}\ \{e\}\ ], \vec{e}_2; n; \sigma) \to_{\mathsf{coop}} (\vec{e}_1, K[()], \vec{e}_2, e; m; \sigma)}$$

$$\frac{\mathit{length}(e_1) = n \qquad 0 \le m < \mathit{length}(\vec{e}_1, K[\mathsf{yield}], \vec{e}_2)}{(\vec{e}_1, K[\mathsf{yield}], \vec{e}_2; n; \sigma) \to_{\mathsf{coop}} (\vec{e}_1, K[()], \vec{e}_2; m; \sigma)}$$

$$\frac{\mathit{length}(e_1) = n}{(\vec{e}_1, K[\mathsf{throw}\ v\ \mathsf{to}\ \mathsf{cont}(K')], \vec{e}_2; n; \sigma) \to_{\mathsf{coop}} (\vec{e}_1, K'[v], \vec{e}_2; n; \sigma)}$$

A quick but careful inspection of the definition of the cooperative step relation above should make it clear that $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ really does capture cooperative concurrency. This is exactly the purpose of defining this programming language: by showing that it can be *correctly* compiled into another programming language by a compiler that compiles the cooperative concurrency primitives using continuations, we establish correctness of an implementation of cooperative concurrency using continuations.

7.2 Translating cooperative concurrency using continuations and its correctness

The translation from $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ into $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ is very straightforward. It simply translates Cfork and yield to programs that use the the light-weight thread library LiThr:[6]

$$\begin{aligned}
\mathsf{LiThr} \triangleq\ & \mathsf{let}\ Q = \mathit{newQueue}\ \mathsf{in} \\
& \mathsf{let}\ \mathit{Frk} = \lambda x.\ \mathsf{call/cc}\ (y.\ \mathit{enqueue}\ Q\ y; \mathsf{throw}\ x\ \mathsf{to}\ (-\ ()))\ \mathsf{in} \\
& \mathsf{let}\ \mathit{Yld} = \lambda_.\ \mathsf{call/cc}\ \begin{pmatrix} x.\ \mathit{enqueue}\ Q\ x; \mathsf{let}\ y = \mathit{dequeue}\ Q\ \mathsf{in} \\ \quad \mathsf{match}\ y\ \mathsf{with}\ \mathit{Some}(z) \Rightarrow \mathsf{throw}\ ()\ \mathsf{to}\ z \mid \mathit{None} \Rightarrow ()\ \mathsf{end} \end{pmatrix}\ \mathsf{in} \\
& (\mathit{Frk}, \mathit{Yld})
\end{aligned}$$

This simple and minimalist light-weight thread library provides two functions: one for forking a new thread, *Frk*, and one for relinquishing control to other threads *Yld*. Notice that the *dequeue* operation can return *None* if the queue empty. However, since in *Yld* the *dequeue* operation is immediately preceded by an *enqueue* operation, this will never happen.

We translate programs in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ intro programs of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ in two phases. First we translate program e of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ into the program $\langle\!\langle e \rangle\!\rangle$ that uses LiThr. In this phase we use $\mathbb{F}$ and $\mathbb{Y}$ as fresh variable names for the names of functions that are provided by LiThr for forking and yielding respectively. The second phase of the translation, $\mathfrak{Comp}(e)$, completes the translation by linking the translation $\langle\!\langle e \rangle\!\rangle$ with LiThr.

The first phase of the translation is as follows:

[6] Note that the specification side, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ does not impose any restriction on scheduling, *i.e.*, yield non-deterministically chooses another thread. Here, on the implementation side we have chosen to use a queue data structure for scheduling of threads.

$$\langle\!\langle x \rangle\!\rangle \triangleq x \qquad \langle\!\langle \mathsf{rec}\, f(x) = e \rangle\!\rangle \triangleq \mathsf{rec}\, f(x) = \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle !\, e \rangle\!\rangle \triangleq !\, \langle\!\langle e \rangle\!\rangle$$
$$\langle\!\langle () \rangle\!\rangle \triangleq () \qquad \langle\!\langle e\ e' \rangle\!\rangle \triangleq \langle\!\langle e \rangle\!\rangle\ \langle\!\langle e' \rangle\!\rangle \qquad \langle\!\langle e \leftarrow e' \rangle\!\rangle \triangleq \Lambda\ \langle\!\langle e \rangle\!\rangle$$
$$\langle\!\langle n \rangle\!\rangle \triangleq n \qquad \langle\!\langle e\ \_ \rangle\!\rangle \triangleq \langle\!\langle e \rangle\!\rangle\ \_ \qquad \langle\!\langle \mathsf{cont}(K) \rangle\!\rangle \triangleq \mathsf{cont}(\langle\!\langle K \rangle\!\rangle)$$
$$\langle\!\langle \Lambda\, e \rangle\!\rangle \triangleq \Lambda\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathsf{fold}\, e \rangle\!\rangle \triangleq \mathsf{fold}\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathsf{call/cc}\,(x.\, e) \rangle\!\rangle \triangleq \mathsf{call/cc}\,(x.\ \langle\!\langle e \rangle\!\rangle)$$
$$\langle\!\langle \mathsf{ref}(e) \rangle\!\rangle \triangleq \mathsf{ref}(\langle\!\langle e \rangle\!\rangle) \qquad \langle\!\langle \mathsf{unfold}\, e \rangle\!\rangle \triangleq \mathsf{unfold}\ \langle\!\langle e \rangle\!\rangle \qquad \langle\!\langle \mathsf{throw}\, e\, \mathsf{to}\, e' \rangle\!\rangle \triangleq \mathsf{throw}\ \langle\!\langle e \rangle\!\rangle\ \mathsf{to}\ \langle\!\langle e' \rangle\!\rangle$$

$$\langle\!\langle \mathsf{yield} \rangle\!\rangle \triangleq \mathbb{Y}\,() \qquad \langle\!\langle \mathsf{Cfork}\ \{e\} \rangle\!\rangle \triangleq \mathbb{F}\,(\lambda\_.\ \langle\!\langle e \rangle\!\rangle)$$

This translation simply turns primitives Cfork and yield into programs that use the interface of LiThr. Notice that the translation of Cfork uses a λ to suspend execution of the translation of e.

Lemma 7.1 (Typing of translation). *Let e be a program in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ such that $\Xi \mid \Gamma \vdash_{\mathsf{coop}} e : \tau$. The following typing judgement holds for the translation of e.*

$$\Xi \mid \Gamma, \mathbb{Y} : 1 \to 1, \mathbb{F} : (1 \to 1) \to 1 \vdash_{\mathsf{seq}} \langle\!\langle e \rangle\!\rangle : \tau$$

The second phase of compilation is defined as follows:

$$\mathfrak{Comp}(e) \triangleq \mathsf{let}\ x = \mathsf{LiThr}\ \mathsf{in}\ \mathsf{let}\ \mathbb{F} = (\pi_1\ x)\ \mathsf{in}\ \mathsf{let}\ \mathbb{Y} = (\pi_2\ x)\ \mathsf{in}\ \langle\!\langle e \rangle\!\rangle$$

Correctness of translation. We will show correctness of our translation by showing observational refinement. To this end, we define propositions $e \Downarrow_{\mathsf{seq}}$ and $e \Downarrow_{\mathsf{coop}}$, which state when programs of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ terminate:

$$e \Downarrow_{\mathsf{seq}} \triangleq \exists v, \sigma.\ (e; \emptyset) \to^*_{\mathsf{seq}} (v; \sigma)$$

and

$$e \Downarrow_{\mathsf{coop}} \triangleq \exists \vec{e}, \sigma.\ (e; 0; \emptyset) \to^*_{\mathsf{coop}} (\vec{e}; n; \sigma) \wedge e_n \text{ is a value}$$

Notice that programs of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ terminate whenever the program reaches a value while programs of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ terminate whenever we reach a state where the current thread n has terminated.

We can now state the following correctness theorem of the translation. Intuitively, the theorem expresses that if the compiled program $\mathfrak{Comp}(e)$ produces a result so would the original program (standard observational refinement).

Theorem 7.2 (Correctness of translation). *Let e be a closed well-typed program in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$, i.e., $\cdot \mid \cdot \vdash_{\mathsf{coop}} e : \tau$. Then we have*

$$\textit{if}\ \mathfrak{Comp}(e) \Downarrow_{\mathsf{seq}} \quad \textit{then}\ e \Downarrow_{\mathsf{coop}}$$

We prove this theorem in the next subsection by setting up a cross-language logical relation between programs of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$. We show that for every $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ program e, the translation $\langle\!\langle e \rangle\!\rangle$ is suitably related to e. Theorem 7.2 then follows by showing that the compiled program $\mathfrak{Comp}(e)$ reduces to $\langle\!\langle e \rangle\!\rangle[\mathit{Frk}, \mathit{Yld}/\mathbb{F}, \mathbb{Y}]$.

7.3 Cross-language logical relation

The types and basic terms of both $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ are the same as $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$. Hence, our cross-language logical relation is very similar to the logical relation presented in Section 3, except that it is defined for pairs of expressions, values and evaluation contexts where the first component pertains to $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and the second to $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$. In fact, the only part of the logical relation in Section 3 that we need to change is the definition of observational refinement. To define the observational refinement relation between $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ we need to introduce resources that we may use for keeping track of the execution on the specification side, *i.e.*, the $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ side. For this purpose, in addition to propositions $\ell \mapsto_s v$ and $j \Rightarrow e$ for keeping track of the heap and the (light-weight) threads on the specification side, we need a proposition to keep track of the current thread on the specification side. Hence, we introduce the proposition $\mathit{CurTh}(j)$, for asserting that the current thread being run is thread j. For all the basic terms of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$, the rules for execution on the specification side remain similar to the ones in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, except that they require the thread being evaluated to be the current thread. As an example, the rule for storing a value in a reference on the specification side is given below. The only rules that change the current thread are those pertaining Cfork and yield.

$$\frac{\ell \mapsto_s v \qquad \mathit{CurTh}(j) \qquad j \Rightarrow K[\ell \leftarrow w]}{\Rrightarrow \mathit{CurTh}(j) * \ell \mapsto_s w * j \Rightarrow K[()]} \qquad \frac{\mathit{CurTh}(j) \qquad j \Rightarrow K[\mathsf{Cfork}\ \{e\}]}{\Rrightarrow \exists j'.\ \mathit{CurTh}(j') * j \Rightarrow K[()] * j' \Rightarrow e}$$

$$\frac{\mathit{CurTh}(j) \qquad j \Rightarrow K[\mathsf{yield}]}{\Rrightarrow \mathit{CurTh}(j) * j \Rightarrow K[()]} \qquad \frac{\mathit{CurTh}(j) \qquad j \Rightarrow K[\mathsf{yield}] \qquad j' \Rightarrow e'}{\Rrightarrow \mathit{CurTh}(j') * j \Rightarrow K[()] * j' \Rightarrow e'}$$

Note that there are two rules pertaining to the execution of yield as it may result in continuing the execution of the same thread.

For our cross-language logical relations we define the observational refinement relation, O^{cross} as follows:

$$O^{\mathrm{cross}}(e, e') \triangleq \forall j.\ \mathit{CurTh}(j) * j \Rightarrow e' \mathrel{-\!\!*} \mathsf{wp}\ e\ \{\exists j', w.\ j' \Rightarrow w * \mathit{CurTh}(j')\}$$

Note how the specification side is expected to be in a thread that has reached a value in its execution, similarly to how we defined $\Downarrow_{\mathrm{coop}}$ above.

7.4 Proof of correctness of translation

In order to prove correctness of the translation we need to reason about a relation between the internal state of the LiThr library and threads on the specification side. In particular, for each continuation K stored in the internal queue of LiThr there must be a thread $j' \Rightarrow e'$ on the specification side such that $K[()]$ observationally refines e'. We use the proposition $\mathit{LiThrInv}$ for this purpose.[7]

$$\mathit{LiThrInv} \triangleq \boxed{\exists l.\ \mathit{isQueue}(Q, l) * \mathop{\Large *}_{K \in l} \exists j', e',\ j' \Rightarrow e' * \square(O^{\mathrm{cross}}(K[()], e'))}^{\mathcal{N}.\mathrm{LiThr}}$$

[7] This is a slight simplification. The proposition $\mathit{LiThrInv}$ is defined using Iris's non-atomic invariants. These are invariants that can be kept open for multiple steps of computation. They are admissible in our system because we have no real concurrency causing racy behavior. It is crucial to be able to keep the $\mathit{LiThrInv}$ open for multiple steps so as to prove that the *dequeue* operation in *Yld* never returns a *None* value. (If the languages had included concurrent racy behaviour, then we could have used a lock in the library and then we would have been able to use standard Iris atomic invariants.) The definition of O^{cross} is also slightly simplified here. It should be slightly adjusted to allow for the use of non-atomic invariants. Iris's weakest preconditions by default only allow atomic invariants.

The proposition $\mathit{isQueue}(Q, l)$ asserts that Q is a queue whose contents are the list l. The two operations of the queue, *enqueue* and *dequeue*, have the following specs:

$$\{\mathit{isQueue}(Q, l)\}^{\mathsf{cl}}$$
$$\quad \mathit{enqueue}\ Q\ v$$
$$\{x.\ x = () * \mathit{isQueue}(Q, v :: l)\}$$

$$\{\mathit{isQueue}(Q, l)\}^{\mathsf{cl}}$$
$$\quad \mathit{dequeue}\ Q$$
$$\{x.\ (\exists l'.\ l = l' +\!\!+ [v] * x = \mathit{Some}(v) * \mathit{isQueue}(Q, l')) \vee (x = \mathit{None} * l = [] * \mathit{isQueue}(Q, l))\}$$

where $+\!\!+$, $::$ and $[\cdot]$ are the usual list operations and [] is the empty list.

To prove the desired correctness theorem (Theorem 7.2), we now prove the fundamental theorem for our cross-language logical relation. It says that, under the assumption that the internal state of the library LiThr is appropriate, then any well-typed $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ program is refined by its translation when linked with the *LightTh* library.

Theorem 7.3 (Fundamental theorem of cross-language logical relations). *Let e be a program in $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ such that $\Xi \mid \Gamma \vdash_{\mathsf{coop}} e : \tau$. Then we have*

$$\mathit{LiThrInv} \Rightarrow \Xi \mid \Gamma \vDash \langle\!\langle e \rangle\!\rangle[\mathit{Frk}, \mathit{Yld}/\mathbb{F}, \mathbb{Y}] \leq_{\mathit{log}} e : \tau$$

Proof. By induction on the derivation of $\Xi \mid \Gamma \vdash_{\mathsf{coop}} e : \tau$. All cases, except for Cfork and yield follow similarly to their counterpart in the proof of Theorem 3.1. Cases Cfork and yield follow by the fact that their translations are applications to $\mathbb{F}$ and $\mathbb{Y}$ respectively under the assumption that the invariant *LiThrInv* holds for the internal queue of LiThr. □

Theorem 7.2 now follows from Theorem 7.3 (in the same way as Theorem 3.2 followed from Theorem 3.1), using and the fact that the program $\mathfrak{Comp}(e)$ reduces to $\langle\!\langle e \rangle\!\rangle[\mathit{Frk}, \mathit{Yld}/\mathbb{F}, \mathbb{Y}]$.

8 MECHANIZATION IN COQ

Taking advantage of the Coq formalization of Iris and Iris Proof Mode (IPM) [Krebbers et al. 2017b], we have mechanized all the technical development and results presented in this paper in Coq. This includes mechanizing the small-step operational semantics of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$, and instantiating Iris with them. Our Coq development is about 15800 lines and includes proofs of contextual refinements for pairs of fine-grained/coarse-grained stacks and counters which we omitted discussion of for reasons of space.

For binders, we use the Autosubst library [Schäfer et al. 2015] which facilitates the use of de Bruijn indices by providing support for simplification of substitutions. In $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$, evaluation contexts are also values and hence also expressions. This forces us to define these mutually inductively. This means that we need to derive the induction principle for these inductive types in Coq by hand. Furthermore, we have to help Autosubst in deriving substitution and simplification lemmas for $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$ that it should otherwise automatically infer. This is mainly why the definition of $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{conc},\mathit{cc}}$, $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc}}$ and $\mathsf{F}^{\mu,\mathit{ref}}_{\mathit{cc},\mathit{coop}}$ combined takes up about 20% of the whole Coq development.

9 RELATED WORK

There has been a considerable body of work on (delimited) continuations, but, we are not aware of any logics or relational models for reasoning about concurrent programs with continuations, let

alone a mechanized framework for relational verification of concurrent programs with continuations.

Program logics for reasoning about continuations. Delbianco and Nanevski [2013] present a type theory for Hoare-style reasoning about an imperative higher-order programming language with (algebraic) continuations, but without concurrency. The system of Delbianco and Nanevski [2013] does not allow higher-order code (including continuations) to be stored in the heap. Note that storing higher-order code in the heap is essential for all our case studies: implementing cooperative concurrency with continuations, implementing the continuation-based web servers and implementing continuations in terms of one-shot continuations. Crolard and Polonowski [2012] develop a program logic for reasoning about jumps but their sequential programming language features no heap or recursive types. Berger [2010] presents a program logic for reasoning about programs in a programming language which is essentially an extension of PCF [Plotkin 1977] with continuations.

Relational reasoning about continuations. The work most closely related to ours is that of Dreyer et al. [2012] who consider a variety of different stateful programming languages and investigate the impact of the higher-order state and control effects (including call/cc and throw). In contrast to our work, they do not consider concurrency. Moreover, they reason directly in a model, whereas we define our logical relation using a program logic (Iris), which means that we can reason more compositionally and at a higher level of abstraction. Another advantage of using Iris, is that we have been able to leverage its Coq formalization and thus to mechanize all of our development. As mentioned in Section 6, our proof that continuations can be expressed in terms of one-shot continuations is inspired by loc. cit.

There are several other works on relational reasoning for sequential programming languages with continuations, *e.g.*, Felleisen and Hieb [1992]; Laird [1997]; Støvring and Lassen [2007]. These differ from our work at least in that they do not consider concurrency.

Relational reasoning about concurrency. There has been much work on relational reasoning about concurrent higher-order imperative programs, without continuations. The work most closely related to ours also is that of Krebbers et al. [2017b], who develop mechanized logical relations (in Iris) for reasoning about contextual equivalence of programs in $\mathsf{F}^{\mu,ref}_{conc}$, a language similar to the one we consider but without call/cc and throw. The approach in *loc. cit.* is based on earlier, non-mechanized logical relations for fine-grained concurrent programs [Birkedal et al. 2012; Turon et al. 2013a,b]. These relational models give an alternative method to linearizability [Herlihy and Wing 1990] for reasoning about contextual refinement for fine-grained concurrent programs. The logical relations method also works in the presence of higher-order programs, which linearizability traditionally struggles with, although there has been some recent promising developments [Cerone et al. 2014; Murawski and Tzevelekos 2017]. In this paper, we have extended the method of logical relations for reasoning about contextual refinement for higher-order fine-grained concurrent programs to work for programs that also use continuations.

10 CONCLUSION AND FUTURE WORK

We have developed a logical relation for $\mathsf{F}^{\mu,ref}_{conc,cc}$, a programming language with advanced features such as impredicative polymorphism à la system F, higher-order mutable references, recursive types, concurrency and most notably continuations. We have devised new non-context-local proof rules for reasoning about weakest preconditions in Iris in the presence of continuations and also introduced context-local weakest preconditions for regaining context-local reasoning about expressions that do not involve non-local control flow. We have defined our relational model and

proved properties thereof in the Iris program logic framework. This has greatly simplified the definition of our relational model, the existence of which is non-trivial because of the type-world circularity [Ahmed 2004; Ahmed et al. 2002; Birkedal et al. 2011]. Furthermore, working inside Iris has enabled us to mechanize the entire development presented in this paper on top of the Coq proof assistant.

We have demonstrated how our logical relation can be used to establish contextual equivalence for a pair of simplified web-server implementations: one storing the state explicitly and one storing the current continuation. The application of context local reasoning in the middle of our logical relatedness proofs demonstrates the usefulness and versatility of context-local weakest preconditions. Finally, we have also given the first (mechanized) proof of the correctness of Friedman and Haynes [1985] encoding of continuations by means of one-shot continuations in a concurrent programming language.

We developed a cross-language logical relation between $\mathsf{F}_{cc}^{\mu,\,ref}$ and $\mathsf{F}_{cc,\,coop}^{\mu,\,ref}$. We used this logical relation to give a compiler-correctness-inspired proof of correctness of the continuation-based implementation of cooperative concurrency. This is to the best of our knowledge the first formal proof of correctness of continuation-based cooperative concurrency.

In the future, we wish to extend our mechanization to reason about delimited continuations [Danvy and Filinski 1990; Felleisen 1988]. Currently our mechanized reasoning is done interactively, in the same style as one reasons in Coq. In the future, we would also like to complement that with more automated reasoning methods.

REFERENCES

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

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*.

Martin Berger. 2010. *Program Logics for Sequential Higher-Order Control.* Springer Berlin Heidelberg, Berlin, Heidelberg, 194–211.

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

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *CSL*.

Andrea Cerone, Alexey Gotsman, and Hongseok Yang. 2014. Parameterised Linearisability. In *ICALP*.

T. Crolard and E. Polonowski. 2012. Deriving a Floyd-Hoare logic for non-local jumps from a formulæ-as-types notion of control. *The Journal of Logic and Algebraic Programming* 81, 3 (2012), 181 – 208. The 22nd Nordic Workshop on Programming Theory (NWPT 2010).

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*. 207–231.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*.

Germán Andrés Delbianco and Aleksandar Nanevski. 2013. Hoare-style reasoning with (algebraic) continuations. *ACM SIGPLAN Notices* 48, 9 (2013), 363–376.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: Compositional Reasoning for Concurrent Programs. In *POPL*.

T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP*. 504–528.

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

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming* 22, 4-5 (2012), 477–528.

Mattias Felleisen. 1988. The Theory and Practice of First-class Prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

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

Matthew Flatt. 2017. More: Systems Programming with Racket. https://docs.racket-lang.org/more/index.html.

Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining Control. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*. ACM, New York, NY, USA, 245–254.

Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. 1984. Continuations and Coroutines *(LFP '84)*.

Greg Hendershott. 2017. http://www.greghendershott.com/2014/09/written-in-racket.html.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990), 463–492.

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, Aleš 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*.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T Graunke, Greg Pettyjohn, and Matthias Felleisen. 2007. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation* 20, 4 (2007), 431–460.

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A Logical Account of a Type-and-Effect System. In *POPL*.

James Laird. 1997. Full Abstraction for Functional Languages with Control. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, Washington, DC, USA, 58–. http://dl.acm.org/citation.cfm?id=788019.788859

Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective Auxiliary State for Coarse-Grained Concurrency. In *POPL*.

Matt Might. 2017. http://matt.might.net/articles/low-level-web-in-racket/.

Andrzej S. Murawski and Nikos Tzevelekos. 2017. Higher-Order Linearisability. In *CONCUR 2017*.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP*.

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

Andrew M. Pitts. 2005. Typed Operational Reasoning. In *Advanced Topics in Types and Programming Languages*, B. C. Pierce (Ed.). The MIT Press, Chapter 7, 245–289.

Gordon D. Plotkin. 1977. LCF considered as a programming language. *Theoretical computer science* 5, 3 (1977), 223–255.

Christian Queinnec. 2004. Continuations and web servers. *Higher-Order and Symbolic Computation* 17, 4 (2004), 277–295.

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

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *PLDI*. 77–87.

Kristian Støvring and Soren Lassen. 2007. A Complete, Co-Inductive Syntactic Theory of Sequential Control and State. In *POPL*.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *ESOP*. 149–168.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A Logical Relation for Monadic Encapsulation of State: Proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018), to appear.

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013a. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*.

Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013b. Logical relations for fine-grained concurrency. In *POPL*.