Mechanized Relational Verification of Concurrent Programs with Continuations: Technical Appendix

Amin Timany Lars Birkedal

November 22, 2017

Contents

1 The language ($\mathsf{F}^{\mu,ref}_{conc,cc}$)

1.1 Statics

$$
\begin{aligned}
e ::= {} & x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{rec}\, f(x) = e \mid e\ e \mid \Lambda\, e \mid e\ \_ \mid \mathsf{fold}\, e \mid \mathsf{unfold}\, e \\
& \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid (e, e) \mid \pi_1\, e \mid \pi_2\, e \\
& \mid \mathsf{inj}_1\, e \mid \mathsf{inj}_1\, e \mid (\mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, x \Rightarrow e\, \mathsf{end}) \\
& \mid \ell \mid \mathsf{ref}(e) \mid\, !\, e \mid e \leftarrow e \mid \mathsf{cas}(e, e, e) \mid \mathsf{fork}\ \{e\} \\
& \mid \mathsf{cont}(K) \mid \mathsf{call/cc}\, (x.\, e) \mid \mathsf{throw}\, e\, \mathsf{to}\, e \\
v ::= {} & () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{rec}\, f(x) = e \mid \Lambda\, e \mid \mathsf{fold}\, v \mid (v, v) \\
& \mid \mathsf{inj}_1\, v \mid \mathsf{inj}_1\, v \mid \ell \mid \mathsf{cont}(K)
\end{aligned}
$$

$$
\begin{aligned}
K ::= {} & - \mid K\ e \mid v\ K \mid K\ \_ \mid \mathsf{fold}\, K \mid \mathsf{unfold}\, K \mid \mathsf{if}\, K\, \mathsf{then}\, e\, \mathsf{else}\, e \mid (K, e) \mid (v, K) \\
& \mid \pi_1\, K \mid \pi_2\, K \mid \mathsf{inj}_1\, K \mid \mathsf{inj}_1\, K \mid (\mathsf{match}\, K\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, x \Rightarrow e\, \mathsf{end}) \\
& \mid \mathsf{ref}(K) \mid\, !\, K \mid K \leftarrow e \mid v \leftarrow K \mid \mathsf{cas}(K, e, e) \mid \mathsf{cas}(v, K, e) \mid \mathsf{cas}(v, v, K) \\
& \mid \mathsf{throw}\, K\, \mathsf{to}\, e \mid \mathsf{throw}\, e\, \mathsf{to}\, K \\
C ::= {} & - \mid \mathsf{rec}\, f(x) = C \mid C\ e \mid e\ C \mid \Lambda\, C \mid C\ \_ \mid \mathsf{fold}\, C \mid \mathsf{unfold}\, C \\
& \mid \mathsf{if}\, C\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, e\, \mathsf{then}\, C\, \mathsf{else}\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, C \\
& \mid (C, e) \mid (e, C) \mid \pi_1\, C \mid \pi_2\, C \\
& \mid \mathsf{inj}_1\, C \mid \mathsf{inj}_1\, C \mid (\mathsf{match}\, C\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, x \Rightarrow e\, \mathsf{end}) \\
& \mid (\mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow C \mid \mathsf{inj}_2\, x \Rightarrow e\, \mathsf{end}) \\
& \mid (\mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e \mid \mathsf{inj}_2\, x \Rightarrow C\, \mathsf{end}) \\
& \mid \mathsf{ref}(C) \mid\, !\, C \mid C \leftarrow e \mid e \leftarrow C \mid \mathsf{cas}(C, e, e) \mid \mathsf{cas}(e, C, e) \mid \mathsf{cas}(e, e, C) \\
& \mid \mathsf{fork}\ \{C\} \mid \mathsf{call/cc}\, (x.\, C) \mid \mathsf{throw}\, C\, \mathsf{to}\, e \mid \mathsf{throw}\, e\, \mathsf{to}\, C
\end{aligned}
$$

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

Notice that K is the set of evaluation contexts while C is the set of all contexts. In fact we have $K \subseteq C$. We write $C[e]$ for filling the hole in context C with e.

Typing

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

$$
\textsc{T-Var}\ \frac{x : \tau \in \Gamma}{\Xi \mid \Gamma \vdash x : \tau}
\qquad
\textsc{T-Unit}\ \ \Xi \mid \Gamma \vdash () : 1
\qquad
\textsc{T-Rec}\ \frac{\Xi \mid \Gamma, x : \tau, f : \tau \to \tau' \vdash e : \tau'}{\Xi \mid \Gamma \vdash \mid \mathsf{rec}\, f(x) = e : \tau \to \tau'}
$$

$$
\textsc{T-App}\ \frac{\Xi \mid \Gamma \vdash e : \tau \to \tau' \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e\ e' : \tau'}
\qquad
\textsc{T-TLam}\ \frac{\Xi, \alpha \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \Lambda\, e : \forall \alpha.\, \tau}
\qquad
\textsc{T-TApp}\ \frac{\Xi \mid \Gamma \vdash e : \forall \alpha.\, \tau}{\Xi \mid \Gamma \vdash e\ \_ : \tau[\tau'/\alpha]}
$$

T-If
$$\frac{\Xi \mid \Gamma \vdash e_1 : \mathbb{B} \qquad \Xi \mid \Gamma \vdash e_2 : \tau \qquad \Xi \mid \Gamma \vdash e_3 : \tau}{\Xi \mid \Gamma \vdash \mathsf{if}\; e_1 \;\mathsf{then}\; e_2 \;\mathsf{else}\; e_3 : \tau}$$

T-Fold
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{fold}\; e : \mu\alpha.\, \tau}$$

T-UnFold
$$\frac{\Xi \mid \Gamma \vdash e : \mu\alpha.\, \tau}{\Xi \mid \Gamma \vdash \mathsf{unfold}\; e : \tau[\mu\alpha.\, \tau/\alpha]}$$

T-Prod
$$\frac{\Xi \mid \Gamma \vdash e : \tau \qquad \Xi \mid \Gamma \vdash e' : \tau'}{\Xi \mid \Gamma \vdash (e, e') : \tau \times \tau'}$$

T-Proj1
$$\frac{\Xi \mid \Gamma \vdash e : \tau \times \tau'}{\Xi \mid \Gamma \vdash \pi_1\, e : \tau}$$

T-Proj2
$$\frac{\Xi \mid \Gamma \vdash e : \tau \times \tau'}{\Xi \mid \Gamma \vdash \pi_2\, e : \tau'}$$

T-Inj$_1$
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{inj}_1\, e : \tau + \tau'}$$

T-Inj2$_2$
$$\frac{\Xi \mid \Gamma \vdash e : \tau'}{\Xi \mid \Gamma \vdash \mathsf{inj}_2\, e : \tau + \tau'}$$

T-Match
$$\frac{\Xi \mid \Gamma \vdash e : \tau + \tau' \qquad \Xi \mid \Gamma, x : \tau \vdash e_1 : \tau \qquad \Xi \mid \Gamma, x : \tau' \vdash e_2 : \tau}{\Xi \mid \Gamma \vdash \ \mid \mathsf{match}\; e \;\mathsf{with}\; \mathsf{inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2 \;\mathsf{end} : \tau}$$

T-Ref
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{ref}(e) : \mathsf{ref}(\tau)}$$

T-DeRef
$$\frac{\Xi \mid \Gamma \vdash e : \mathsf{ref}(\tau)}{\Xi \mid \Gamma \vdash {!}\, e : \tau}$$

T-Assign
$$\frac{\Xi \mid \Gamma \vdash e : \mathsf{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e' : \tau}{\Xi \mid \Gamma \vdash e \leftarrow e' : 1}$$

T-CAS
$$\frac{\Xi \mid \Gamma \vdash e_1 : \mathsf{ref}(\tau) \qquad \Xi \mid \Gamma \vdash e_2 : \tau \qquad \Xi \mid \Gamma \vdash e_3 : \tau}{\Xi \mid \Gamma \vdash \mathsf{cas}(e_1, e_2, e_3) : 1}$$

T-Fork
$$\frac{\Xi \mid \Gamma \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{fork}\; \{e\} : 1}$$

T-Cont
$$\frac{K : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi \mid \Gamma; \tau')}{\Xi \mid \Gamma \vdash \mathsf{cont}(K) : \mathsf{cont}(\tau)}$$

T-Call/cc
$$\frac{\Xi \mid \Gamma, x : \mathsf{cont}(\tau) \vdash e : \tau}{\Xi \mid \Gamma \vdash \mathsf{call/cc}\, (x.\, e) : \tau}$$

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

Context typing

$$\boxed{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

CT-Hole
$$- : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi \mid \Gamma; \tau)$$

CT-Rec
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma', x : \tau', f : \tau' \to \tau''; \tau'')}{\mathsf{rec}\; f(x) = C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' \to \tau'')}$$

CT-App$_1$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau') \qquad \Xi \mid \Gamma \vdash e : \tau' \to \tau''}{e\; C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}$$

CT-App$_2$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' \to \tau'') \qquad \Xi' \mid \Gamma' \vdash e : \tau'}{C\ e : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}$$

CT-TLam
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi', \alpha \mid \Gamma'; \tau')}{\Lambda\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \forall \alpha.\, \tau')}$$

CT-TApp
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \forall \alpha.\, \tau')}{C\ \_ : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'[\tau''/\alpha])}$$

CT-Fold
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{\mathsf{fold}\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mu\alpha.\, \tau')}$$

CT-UnFold
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mu\alpha.\, \tau')}{\mathsf{unfold}\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'[\mu\alpha.\, \tau'/\alpha])}$$

CT-If$_1$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mathbb{B}) \qquad \Xi' \mid \Gamma' \vdash e_2 : \tau' \qquad \Xi' \mid \Gamma' \vdash e_3 : \tau'}{\mathsf{if}\, C\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

CT-If$_2$
$$\frac{\Xi' \mid \Gamma' \vdash e_1 : \mathbb{B} \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau') \qquad \Xi' \mid \Gamma' \vdash e_3 : \tau'}{\mathsf{if}\, e_1\, \mathsf{then}\, C\, \mathsf{else}\, e_3 : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

CT-If$_3$
$$\frac{\Xi' \mid \Gamma' \vdash e_1 : \mathbb{B} \qquad \Xi' \mid \Gamma' \vdash e_2 : \tau' \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

CT-Prod$_1$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau') \qquad \Xi' \mid \Gamma' \vdash e : \tau''}{(C, e) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' \times \tau'')}$$

CT-Proj$_1$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' \times \tau'')}{\pi_1\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

CT-Prod$_2$
$$\frac{\Xi' \mid \Gamma' \vdash e : \tau' \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}{(e, C) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' \times \tau'')}$$

CT-Proj$_2$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' \times \tau'')}{\pi_2\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}$$

CT-Inj$_1$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{\mathsf{inj}_1\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' + \tau'')}$$

CT-Inj$_2$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}{\mathsf{inj}_2\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' + \tau'')}$$

CT-Match$_1$
$$\frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau' + \tau'') \qquad \Xi' \mid \Gamma', x : \tau' \vdash e_1 : \tau_2 \qquad \Xi' \mid \Gamma', x : \tau'' \vdash e_2 : \tau_2}{\mathsf{match}\, C\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2\, \mathsf{end} : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau_2)}$$

CT-Match$_2$
$$\frac{\Xi' \mid \Gamma' \vdash e : \tau' + \tau'' \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma', x : \tau'; \tau_2) \qquad \Xi' \mid \Gamma', x : \tau'' \vdash e_2 : \tau_2}{\mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow C \mid \mathsf{inj}_2\, x \Rightarrow e_2\, \mathsf{end} : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau_2)}$$

$$\textsc{CT-Match}_3\quad \frac{\Xi' \mid \Gamma' \vdash e : \tau' + \tau'' \qquad \Xi' \mid \Gamma', x : \tau' \vdash e_1 : \tau_2 \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma', x : \tau''; \tau_2)}{\mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow C\, \mathsf{end} : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau_2)}$$

$$\textsc{CT-Ref}\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{\mathsf{ref}(C) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mathsf{ref}(\tau'))} \qquad \textsc{CT-DeRef}\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mathsf{ref}(\tau'))}{!\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

$$\textsc{CT-Assign}_1\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mathsf{ref}(\tau')) \qquad \Xi' \mid \Gamma' \vdash e : \tau'}{C \leftarrow e : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; 1)}$$

$$\textsc{CT-Assign}_2\quad \frac{\Xi' \mid \Gamma' \vdash e : \mathsf{ref}(\tau') \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{e \leftarrow C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; 1)}$$

$$\textsc{CT-CAS}_1\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mathsf{ref}(\tau')) \qquad \Xi' \mid \Gamma' \vdash e_2 : \tau' \qquad \Xi' \mid \Gamma' \vdash e_3 : \tau'}{\mathsf{cas}(C, e_2, e_3) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; 1)}$$

$$\textsc{CT-CAS}_2\quad \frac{\Xi' \mid \Gamma' \vdash e_1 : \mathsf{ref}(\tau') \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau') \qquad \Xi' \mid \Gamma' \vdash e_3 : \tau'}{\mathsf{cas}(e_1, C, e_3) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; 1)}$$

$$\textsc{CT-CAS}_3\quad \frac{\Xi' \mid \Gamma' \vdash e_1 : \mathsf{ref}(\tau') \qquad \Xi' \mid \Gamma' \vdash e_2 : \tau' \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{\mathsf{cas}(e_1, e_2, C) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; 1)}$$

$$\textsc{CT-Fork}\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}{\mathsf{fork}\ \{C\} : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; 1)} \qquad \textsc{CT-Call/cc}\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma', x : \mathsf{cont}(\tau'); \tau')}{\mathsf{call/cc}\, (x.\, C) : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')}$$

$$\textsc{CT-Throw}_1\quad \frac{C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau') \qquad \Xi' \mid \Gamma' \vdash e : \mathsf{cont}(\tau')}{\mathsf{throw}\, C\, \mathsf{to}\, e : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}$$

$$\textsc{CT-Throw}_2\quad \frac{\Xi' \mid \Gamma' \vdash e : \tau' \qquad C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \mathsf{cont}(\tau'))}{\mathsf{throw}\, e\, \mathsf{to}\, C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau'')}$$

Lemma 1. *Let C be a context such that $C : (\Xi \mid \Gamma; \tau) \rightsquigarrow (\Xi' \mid \Gamma'; \tau')$ then for any term e such that $\Xi \mid \Gamma \vdash e : \tau$ we have $\Xi' \mid \Gamma' \vdash C[e] : \tau'$.*

1.2 Dynamics

Evaluation-context aware head reduction

$$\boxed{(e,\sigma) \to_K (e',\sigma')}$$

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

$$(\mathsf{unfold}\,(\mathsf{fold}\, v), \sigma) \to_K (v, \sigma) \qquad (\mathsf{if\ true\ then}\, e_2\, \mathsf{else}\, e_3, \sigma) \to_K (e_2, \sigma)$$

$$(\mathsf{if\ false\ then}\, e_2\, \mathsf{else}\, e_3, \sigma) \to_K (e_3, \sigma) \qquad (\pi_1\,(v_1, v_2), \sigma) \to_K (v_1, \sigma)$$

$$(\pi_2\,(v_1, v_2), \sigma) \to_K (v_2, \sigma)$$

$$(\mathsf{match\ inj}_1\, v\, \mathsf{with\ inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2\, \mathsf{end}, \sigma) \to_K (e_1[v/x], \sigma)$$

$$(\mathsf{match\ inj}_2\, v\, \mathsf{with\ inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2\, \mathsf{end}, \sigma) \to_K (e_2[v/x], \sigma)$$

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

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

$$\frac{\sigma = \sigma' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\mathsf{cas}(\ell, v, v'), \sigma) \to_K (\mathsf{false}, \sigma)} \qquad (\mathsf{call/cc}\,(x.\, e), \sigma) \to_K (e[\mathsf{cont}(K)/x], \sigma)$$

Thread-pool reduction

$$\boxed{(\vec{e}_1; \sigma) \to (\vec{e}_2; \sigma')}$$

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

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

Definition 2 (Context refinement). *Let e and e' be two terms. Then contextual refinement of e' by e at type $(\Xi; \Gamma; \tau)$, written as $\Xi \mid \Gamma \vdash e \leq_{ctx} e' : \tau$ is defined as follows.*

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

where

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

Definition 3 (Context equivalence)**.** *Let e and e' be two terms. Then contextual equivalence of e' and e at type $(\Xi; \Gamma; \tau)$, written as $\Xi \mid \Gamma \vdash e \approx_{ctx} e' : \tau$ is defined as follows.*

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

2 An Iris primer

Iris Jung et al. [2015, 2016], 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:

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

Here *Ectx*, *Var*, *Expr* and *Val* are Iris types for evaluation contexts, variables, expressions and values of $\mathsf{F}^{\mu,ref}_{conc,cc}$. 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:

$$\begin{aligned} P ::= {} & \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \\ & \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi \mid \triangleright P \mid \mu r.P \mid \Box P \\ & \mid \mathsf{wp}\, e \left\{x.\ P\right\} \mid \{P\}\, e\, \{x.\ Q\} \mid {\Rrightarrow} P \mid \boxed{P}^{\mathcal{N}} \\ & \mid \checkmark(a) \mid \boxed{a : \mathcal{M}}^{\gamma} \mid \ldots \end{aligned}$$

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 owned 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 $\mathsf{F}^{\mu,ref}_{conc,cc}$. 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 $\mathsf{F}^{\mu,ref}_{conc,cc}$ 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, then those relations should be duplicable. We use the persistence modality $\square$ to ensure this.

Iris facilitates specification and verification of programs by means of weakest-preconditions $\mathsf{wp}\, e \left\{v.\, \Phi\right\}$, 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 $\mathsf{wp}\, e \left\{x.\, P\right\}$ as $\mathsf{wp}\, e \left\{P\right\}$. Also, we sometimes write $\mathsf{wp}\, e \left\{\Phi\right\}$ for $\mathsf{wp}\, e \left\{x.\, \Phi(x)\right\}$

In Iris, Hoare triples are defined in terms of weakest preconditions like this:

$$\{P\}\, e\, \{v.\, Q\} \triangleq \square \left(P \twoheadrightarrow \mathsf{wp}\, e \left\{v.\, Q\right\}\right)$$

Note that the $\square$ 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).

Note that 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 langauges 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

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

Clearly, this rule is not sound when expressions include $\mathsf{call/cc}$ (since $\mathsf{call/cc}$ captures the context its behaviour depends on the context).

Thus, for reasoning about $\mathsf{F}^{\mu,ref}_{conc,cc}$ we cannot use the “standard” Iris rules Jung et al. [2015, 2016], Krebbers et al. [2017a,b] for weakest preconditions. Instead, we use new rules such as the following — 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 so we cannot do that). Note that the context is used in the rules CALLCC-WP and THROW-WP for $\mathsf{call/cc}$ and throw. These two rules directly reflect the operational semantics of $\mathsf{call/cc}$ and throw.

FST-WP
$$\frac{\triangleright \mathsf{wp}\, K[v]\, \{\Phi\}}{\mathsf{wp}\, K[\pi_1\,(v, w)]\, \{\Phi\}}$$

IF-TRUE-WP
$$\frac{\triangleright \mathsf{wp}\, K[e]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{if\ true\ then}\ e\ \mathsf{else}\ e']\, \{\Phi\}}$$

REC-WP
$$\frac{\triangleright \mathsf{wp}\, K[e[\mathsf{rec}\, f(x) = e, v/f, x]]\, \{\Phi\}}{\mathsf{wp}\, K[(\mathsf{rec}\, f(x) = e)\ v]\, \{\Phi\}}$$

CALLCC-WP
$$\frac{\triangleright \mathsf{wp}\, K[e[\mathsf{cont}(K)/x]]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{call/cc}\,(x.\, e)]\, \{\Phi\}}$$

THROW-WP
$$\frac{\triangleright \mathsf{wp}\, K'[v]\, \{\Phi\}}{\mathsf{wp}\, K[\mathsf{throw}\ v\ \mathsf{to}\ \mathsf{cont}(K')]\, \{\Phi\}}$$

In summa, for $\mathsf{F}^{\mu,ref}_{conc,cc}$ *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 5 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 are for expressions will perform 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 also idempotent, $\Rrightarrow\Rrightarrow P \dashv\vdash \Rrightarrow P$. We write $P \Rrightarrow\!\!* \; Q$ as shorthand for $P \mathrel{-\!\!*} \Rrightarrow Q$. Crucially, resources can be updated throughout a proof of weakest preconditions:

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

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.

INV-ALLOC
$$\frac{\triangleright P}{\Rrightarrow \boxed{P}^{\mathcal{N}}}$$

INV-OPEN-WP
$$\frac{\begin{array}{c} e \text{ is atomic} \\ \boxed{R}^{\mathcal{N}} \qquad (\triangleright R) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{y.\ (\triangleright R) * \mathsf{wp}\, K[y]\, \{x.\ Q\}\}\end{array}}{\mathsf{wp}\, K[e]\, \{x.\ Q\}}$$

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

[2] Ofiicially 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.

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.*, $\boxed{P}^{\mathcal{N}} \dashv\vdash \Box\boxed{P}^{\mathcal{N}}$.

2.1 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.
- $j \Mapsto e$: the thread j on the specification is abotu to execute e.

These resource 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. See Sections 3 and 4. In the reset of this section we describe resources in Iris base logic and how these can be used to define the resources above for weakest preconditions and executions on the specification side.

2.2 Ghost State

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

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

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

Some Useful Monoids In this paragraph, we describe a few monoids which are particularly useful and which we will use in the following. We do not give the full definitions of the monoids (those can be found in [Krebbers et al., 2017a]), but focus instead on the

Auth-Included
$\bullet\, a \cdot \circ\, b \vdash b \subseteq a$

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

Agreement-Valid
$\checkmark(\mathsf{ag}(a) \cdot \mathsf{ag}(b)) \dashv\vdash a = b$

Exclusive
$\cancel{\checkmark}(\mathsf{ex}(a) \cdot b)$

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

Full-Exclusive
$\cancel{\checkmark}(\bullet\, a \cdot \bullet\, b)$

Finset-op
$a \cdot b = a \cup b$

Auth-Alloc-Finset
$\boxed{\bullet\, h}^{\gamma} \Rrightarrow_{*} \boxed{\bullet\, (h \cup a) \cdot \circ\, a}^{\gamma}$

Auth-Finset-valid
$\checkmark(\bullet\, h \cdot \circ\, a) \dashv\vdash a \subseteq h$

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

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

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

Auth-Update-Fpfn
$$\boxed{\bullet\, (h \uplus (\ell \mapsto \mathsf{ex}(v_1))) \cdot \circ\, \ell \mapsto \mathsf{ex}(v_1)}^{\gamma} \Rrightarrow_{*} \boxed{\bullet\, (h \uplus (\ell \mapsto \mathsf{ex}(v_2))) \cdot \circ\, \ell \mapsto \mathsf{ex}(v_2)}^{\gamma}$$

Figure 1: Rules for selected monoid resources in Iris

properties which the elements of the monoids satisfy, shown in Figure 1. These rules stated are only for monoids that we use in this work and not in Iris in its generality. For instance, in the rule Auth-Included, $\subseteq$ is a set relation and is defined for finite set and finite partial function monoids and not in general.

The figure depicts the rules necessary for allocating and updating finite set monoids, $\mathsf{finset}(A)$, and finite partial function monoids, $A \xrightharpoonup{\text{fin}} M$. In the finset monoid, the operation $\cdot$ is union, $\cup$. Hence, $\mathsf{finset}(A)$ is persistent:

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

In finite map monoid, the monoid operation $x \cdot y$ is *disjoint* union. The notation $a \mapsto b : A \xrightharpoonup{\text{fin}} B \triangleq \{(a, b)\}$ is a singleton finite partial function.

The constructs $\bullet$ and $\circ$ are constructors of the so-called authoritative monoid $\textsc{Auth}(M)$. We read $\bullet\, a$ as *full* a and $\circ\, a$ as *fragment* a. We use the authoritative monoid to distribute ownership of fragments of a resource. The intuition is that $\bullet\, a$ is the authoritative knowledge of the full resource, think of it as being kept track of in a central location. This central location is the full part of the resource (see rule Auth-Included). The fragments, $\circ\, a$, can be shared (rule Frag-distributes) while the full part (the central location) should always remain unique (rule Full-Exclusive).

In addition to authoritative monoids, we also use the agreement monoid $\textsc{Ag}(M)$ and exclusive monoid $\textsc{Ex}(M)$. As the name suggests, the operation of the agreement monoid guarantees that $\mathsf{ag}(a) \cdot \mathsf{ag}(b)$ is invalid whenever $a \neq b$ (and otherwise it is idempotent; see rules Agree and Agreement-Valid). From the rule Agree it follows that the ownership of elements of $\textsc{Ag}(M)$ is persistent.

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

Similarly, ownership of authoritative fragments of finite sets is also persistent.

$$\boxed{\circ\,(a)}^{\gamma} \dashv\vdash \boxed{\circ\,(a\cdot a)}^{\gamma} \dashv\vdash \boxed{\circ\,(a)\cdot\circ\,(a)}^{\gamma} \dashv\vdash \boxed{\circ\,(a)}^{\gamma} * \boxed{\circ\,(a)}^{\gamma}$$

The operation of the exclusive monoid never results in a valid element (rule Exclusive), enforcing that there can only be one instance of it owned.

2.3 Using Ghost State to Represent Resources used in defining logical relations

We can now give meaning to the heap-specific predicates used in the in weakest-preconditions and in the specification side execution rules, by presenting the canonical example of a Heap monoid:

$$\textsc{Heap} \triangleq \textsc{Auth}(Loc \overset{\text{fin}}{\rightharpoonup} (\textsc{Ex}(Val))) \quad \ell \mapsto_i v \triangleq \boxed{\circ\,[l \mapsto \mathsf{ex}(v)]}^{\gamma_i} \quad \ell \mapsto_s v \triangleq \boxed{\circ\,[l \mapsto \mathsf{ex}(v)]}^{\gamma_s}$$

Where γ_i and γ_s are arbitrary but globally fixed names for resources keeping track heaps of the implementation and specification sides. Notice here that Heap is build from nesting Ex in the finite partial functions monoid, which again is nested in the Auth monoid. Therefore, to allocate and update and in the Heap monoid, we can use Auth-Alloc-Fpfn and Auth-Update-Fpfn respectively.

Now the definition of weakest precondition for an expression e, $\mathsf{wp}\, e \left\{\Phi\right\}$ can be defined roughly as:[3]

$$\begin{aligned}\mathsf{wp}\, e \left\{\Phi\right\} \triangleq \mu R.\ & (e \in Val * \mathrel{|\!\!\Rrightarrow}\Phi(e)) \lor \\ & (\forall \sigma.\ \boxed{\bullet\,\sigma}^{\gamma_i} \mathrel{-\!\!*} reducible(\sigma, e) * \triangleright \forall e', \sigma'.\ (\sigma, e) \to (\sigma', e') \mathrel{\equiv\!\!*} \boxed{\bullet\,\sigma'}^{\gamma_i} * R(e', \Phi))\end{aligned}$$

Note that this definition is defined using Iris' internal fixpoint combinator. This is in fact how logical steps are tied to operational steps. This definition basically says that evaluation e under the heap that we currently own is safe: it is either a value for which we know the post condition holds or it is reducible to something for which the weakest precondition holds.

For the specification side thread pool we use a finite partial map mapping from thread identifiers to expressions.

$$\textsc{Threadpool} \triangleq \textsc{Auth}(\mathbb{N} \overset{\text{fin}}{\rightharpoonup} (\textsc{Ex}(Expr))) \quad j \Mapsto e \triangleq \boxed{\circ\,[j \mapsto \mathsf{ex}(e)]}^{\gamma_{tp}}$$

Here, the ghost name γ_{tp} is a globally fixed name for the thread-pool resource. Tying the expression heap for the and specification side is done by means of an invariant:

[3]The actual definition of weakest preconditions in Krebbers et al. [2017a] is more involved but this very simplified version suffices for our discussions.

$$\textsc{specInv}(\rho) \triangleq \boxed{\exists \vec{e}, \sigma.\ \left[\bullet\, \sigma\right]^{\gamma_s} * \left[\bullet\, \mathsf{fpfnOf}(\vec{e})\right]^{\gamma_{tp}} * \rho \to^* (\sigma, \vec{e})}^{\mathcal{N}.tp}$$

Here fpfnOf is a function that given a sequence of expressions (threads running) returns a finite partial function representing it:

$$\mathsf{fpfnOf}(e_0, \ldots, e_n) \triangleq (0 \mapsto \mathsf{ex}(e_0)) \cup \cdots \cup (n \mapsto \mathsf{ex}(e_n))$$

The configuration ρ (a pair of a heap and a thread pool) is to be though of as an initial point of execution. That is, this invariant states intuitively that, we are in a configuration owned by resources such that this configuration is reachable from our starting point ρ. All our lemmas and rules about execution on the specification side implicitly assume this invariant universally quantifying ρ. This means that proving a rule like UNFOLD-STEP boils down to proving that given any starting configuration ρ if we can reach a configuration where under thread j we have $K[\mathsf{unfold}\ (\mathsf{fold}\ v)]$ then we can also reach a configuration from ρ that under thread j we have $K[v]$. Note that if this is the case we can reestablish the invariant by simply updating the thread-pool resource (in this case we need not update the specification side heap!).

3 Weakest precondition rules

REC-WP
$$\frac{\triangleright \mathsf{wp}\ K[e[\mathsf{rec}\ f(x) = e, v/f, x]]\ \{\Phi\}}{\mathsf{wp}\ K[(\mathsf{rec}\ f(x) = e)\ v]\ \{\Phi\}}$$

TAPP-WP
$$\frac{\triangleright \mathsf{wp}\ K[e]\ \{\Phi\}}{\mathsf{wp}\ K[(\Lambda\, e)\ \_]\ \{\Phi\}}$$

UNFOLD-WP
$$\frac{\triangleright \mathsf{wp}\ K[v]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{unfold}\ (\mathsf{fold}\ v)]\ \{\Phi\}}$$

IF-TRUE-WP
$$\frac{\triangleright \mathsf{wp}\ K[e]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{if\ true\ then}\ e\ \mathsf{else}\ e']\ \{\Phi\}}$$

IF-FALSE-WP
$$\frac{\triangleright \mathsf{wp}\ K[e']\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{if\ false\ then}\ e\ \mathsf{else}\ e']\ \{\Phi\}}$$

FST-WP
$$\frac{\triangleright \mathsf{wp}\ K[v]\ \{\Phi\}}{\mathsf{wp}\ K[\pi_1\ (v, w)]\ \{\Phi\}}$$

SND-WP
$$\frac{\triangleright \mathsf{wp}\ K[w]\ \{\Phi\}}{\mathsf{wp}\ K[\pi_2\ (v, w)]\ \{\Phi\}}$$

MATCH-INJ1-WP
$$\frac{\triangleright \mathsf{wp}\ K[e_1[v/x]]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{match\ inj}_1\, v\ \mathsf{with\ inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2\ \mathsf{end}]\ \{\Phi\}}$$

MATCH-INJ2-WP
$$\frac{\triangleright \mathsf{wp}\ K[e_2[v/x]]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{match\ inj}_2\, v\ \mathsf{with\ inj}_1\, x \Rightarrow e_1 \mid \mathsf{inj}_2\, x \Rightarrow e_2\ \mathsf{end}]\ \{\Phi\}}$$

ALLOC-WP
$$\frac{\forall \ell.\ \ell \mapsto_i v \twoheadrightarrow \mathsf{wp}\ K[\ell]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{ref}(v)]\ \{\Phi\}}$$

LOAD-WP
$$\frac{\ell \mapsto_i v \twoheadrightarrow \mathsf{wp}\ K[v]\ \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\ K[!\,\ell]\ \{\Phi\}}$$

STORE-WP
$$\frac{\ell \mapsto_i w \twoheadrightarrow \triangleright \mathsf{wp}\ K[()]\ \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\ K[\ell \leftarrow w]\ \{\Phi\}}$$

CAS-SUC-WP
$$\frac{\ell \mapsto_i w \twoheadrightarrow \triangleright \mathsf{wp}\ K[\mathsf{true}]\ \{\Phi\} \qquad \triangleright\, \ell \mapsto_i v}{\mathsf{wp}\ K[\mathsf{cas}(\ell, v, w)]\ \{\Phi\}}$$

CAS-FAIL-WP
$$\frac{\ell \mapsto_i v' \mathrel{-\!\!*} \triangleright \mathsf{wp}\ K[\mathsf{false}]\ \{\Phi\} \qquad \triangleright \ell \mapsto_i v' \qquad v \neq v'}{\mathsf{wp}\ K[\mathsf{cas}(\ell, v, w)]\ \{\Phi\}}$$

FORK-WP
$$\frac{\triangleright \mathsf{wp}\ K[()]\ \{\Phi\} \qquad \mathsf{wp}\ e\ \{\_.\ \top\}}{\mathsf{wp}\ K[\mathsf{fork}\ \{e\}]\ \{\Phi\}}$$

CALLCC-WP
$$\frac{\triangleright \mathsf{wp}\ K[e[\mathsf{cont}(K)/x]]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{call/cc}\ (x.\ e)]\ \{\Phi\}}$$

THROW-WP
$$\frac{\triangleright \mathsf{wp}\ K'[v]\ \{\Phi\}}{\mathsf{wp}\ K[\mathsf{throw}\ v\ \mathsf{to}\ \mathsf{cont}(K')]\ \{\Phi\}}$$

4 Rules for execution on the specification side

REC-STEP
$$\frac{j \Mapsto K[\mathsf{rec}\ f(x) = e]}{\Rrightarrow j \Mapsto K[e[\mathsf{rec}\ f(x) = e, v/f, x]]}$$

TAPP-STEP
$$\frac{j \Mapsto K[(\Lambda\, e)\ \_]}{\Rrightarrow j \Mapsto K[e]}$$

UNFOLD-STEP
$$\frac{j \Mapsto K[\mathsf{unfold}\ (\mathsf{fold}\ v)]}{\Rrightarrow j \Mapsto K[v]}$$

IF-TRUE-STEP
$$\frac{j \Mapsto K[\mathsf{if\ true\ then}\ e\ \mathsf{else}\ e']}{\Rrightarrow j \Mapsto K[e]}$$

IF-FALSE-STEP
$$\frac{j \Mapsto K[\mathsf{if\ false\ then}\ e\ \mathsf{else}\ e']}{\Rrightarrow j \Mapsto K[e']}$$

FST-STEP
$$\frac{j \Mapsto K[\pi_1\ (v, w)]}{\Rrightarrow j \Mapsto K[v]}$$

SND-STEP
$$\frac{j \Mapsto K[\pi_2\ (v, w)]}{\Rrightarrow j \Mapsto K[w]}$$

MATCH-INJ1-STEP
$$\frac{j \Mapsto K[\mathsf{match\ inj}_1\ v\ \mathsf{with\ inj}_1\ x \Rightarrow e_1 \mid \mathsf{inj}_2\ x \Rightarrow e_2\ \mathsf{end}]}{\Rrightarrow j \Mapsto K[e_1[v/x]]}$$

MATCH-INJ2-STEP
$$\frac{j \Mapsto K[\mathsf{match\ inj}_2\ v\ \mathsf{with\ inj}_1\ x \Rightarrow e_1 \mid \mathsf{inj}_2\ x \Rightarrow e_2\ \mathsf{end}]}{\Rrightarrow j \Mapsto K[e_2[v/x]]}$$

ALLOC-STEP
$$\frac{j \Mapsto K[\mathsf{ref}(v)]}{\Rrightarrow \exists \ell.\ \ell \mapsto_s v * j \Mapsto K[\ell]}$$

LOAD-STEP
$$\frac{\ell \mapsto_s v \qquad j \Mapsto K[!\,\ell]}{\Rrightarrow \exists \ell.\ \ell \mapsto_s v * j \Mapsto K[v]}$$

STORE-STEP
$$\frac{\ell \mapsto_s v \qquad j \Mapsto K[\ell \leftarrow w]}{\Rrightarrow \ell \mapsto_s w * j \Mapsto K[()]}$$

CAS-SUC-STEP
$$\frac{\ell \mapsto_s v \qquad j \Mapsto K[\mathsf{cas}(\ell, v, w)]}{\Rrightarrow \ell \mapsto_s w * j \Mapsto K[\mathsf{true}]}$$

CAS-FAIL-STEP
$$\frac{\ell \mapsto_s v' \qquad j \Mapsto K[\mathsf{cas}(\ell, v, w)] \qquad v \neq v'}{\Rrightarrow \ell \mapsto_s v' * j \Mapsto K[\mathsf{false}]}$$

FORK-STEP
$$\frac{j \Mapsto K[\mathsf{fork}\ \{e\}]}{\Rrightarrow j \Mapsto K[()] * \exists j'.\ j' \Mapsto e}$$

CALLCC-STEP
$$\frac{j \Mapsto K[\mathsf{call/cc}\ (x.\ e)]}{\Rrightarrow j \Mapsto K[e[\mathsf{cont}(K)/x]]}$$

THROW-STEP
$$\frac{j \Mapsto K[\mathsf{throw}\ v\ \mathsf{to}\ K']}{\Rrightarrow j \Mapsto K'[v]}$$

5 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. *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. These are presented in Section 6.

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, then 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 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:

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

6 Context-local Weakest precondition rules

REC-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e[\mathsf{rec}\ f(x) = e, v/f, x]\ \{\Phi\}}{\mathsf{clwp}\ (\mathsf{rec}\ f(x) = e)\ v\ \{\Phi\}}$$

TAPP-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e\ \{\Phi\}}{\mathsf{clwp}\ (\Lambda\, e)\ \_\ \{\Phi\}}$$

UNFOLD-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ v\ \{\Phi\}}{\mathsf{clwp}\ \mathsf{unfold}\ (\mathsf{fold}\ v)\ \{\Phi\}}$$

IF-TRUE-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e\ \{\Phi\}}{\mathsf{clwp}\ K[\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e\ \mathsf{else}\ e']\ \{\Phi\}}$$

IF-FALSE-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e'\ \{\Phi\}}{\mathsf{clwp}\ \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e\ \mathsf{else}\ e'\ \{\Phi\}}$$

FST-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ v\ \{\Phi\}}{\mathsf{clwp}\ \pi_1\ (v, w)\ \{\Phi\}}$$

SND-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ w\ \{\Phi\}}{\mathsf{clwp}\ \pi_2\ (v, w)\ \{\Phi\}}$$

MATCH-INJ1-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e_1[v/x]\ \{\Phi\}}{\mathsf{clwp}\ \mathsf{match}\ \mathsf{inj}_1\ v\ \mathsf{with}\ \mathsf{inj}_1\ x \Rightarrow e_1 \mid \mathsf{inj}_2\ x \Rightarrow e_2\ \mathsf{end}\ \{\Phi\}}$$

MATCH-INJ2-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e_2[v/x]\ \{\Phi\}}{\mathsf{clwp}\ \mathsf{match}\ \mathsf{inj}_2\ v\ \mathsf{with}\ \mathsf{inj}_1\ x \Rightarrow e_1 \mid \mathsf{inj}_2\ x \Rightarrow e_2\ \mathsf{end}\ \{\Phi\}}$$

ALLOC-CLWP
$$\mathsf{clwp}\ K[\mathsf{ref}(v)]\ \{v.\ \exists \ell.\ v = \ell * \ell \mapsto_i v\}$$

LOAD-CLWP
$$\frac{\triangleright\, \ell \mapsto_i v}{\mathsf{wp}\ !\,\ell\ \{w.\ w = v * \ell \mapsto_i v\}}$$

STORE-CLWP
$$\frac{\triangleright\, \ell \mapsto_i v}{\mathsf{wp}\ \ell \leftarrow w\ \{v'.\ v' = () * \ell \mapsto_i w\}}$$

CAS-SUC-WP
$$\frac{\triangleright\, \ell \mapsto_i v}{\mathsf{wp}\ \mathsf{cas}(\ell, v, w)\ \{v'.\ v\ = \mathsf{true} * \ell \mapsto_i w\}}$$

CAS-FAIL-CLWP
$$\frac{\triangleright\, \ell \mapsto_i v' \qquad v \neq v'}{\mathsf{wp}\ \mathsf{cas}(\ell, v, w)\ \{v''.\ v'' = \mathsf{false} * \ell \mapsto_i v'\}}$$

FORK-CLWP
$$\frac{\triangleright\, \mathsf{clwp}\ e\ \{\_.\ \top\}}{\mathsf{clwp}\ \mathsf{fork}\ \{e\}\ \{v.\ v = ()\}}$$

7 Logical Relations

Observational refinement ($\mathcal{O} : \mathit{Expr} \times \mathit{Expr} \to \mathit{iProp}$):

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

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

$$\begin{aligned}
[\![\Xi \vdash \alpha]\!]_\Delta &\triangleq \Delta(\alpha)\\
[\![\Xi \vdash 1]\!]_\Delta(v, v') &\triangleq v = v' = ()\\
[\![\Xi \vdash \mathbb{B}]\!]_\Delta(v) &\triangleq v = v' = \mathsf{true} \lor v = v' = \mathsf{false}\\
[\![\Xi \vdash \mathbb{N}]\!]_\Delta(v, v') &\triangleq \exists n.\ v = v' = n\\
[\![\Xi \vdash \tau_1 + \tau_2]\!]_\Delta(v, v') &\triangleq \bigvee_{i \in \{1,2\}} (\exists w, w'.\ v = \mathsf{inj}_i\, w \land v' = \mathsf{inj}_i\, w' \land [\![\Xi \vdash \tau_i]\!]_\Delta(w, w'))\\
[\![\Xi \vdash \tau \times \tau']\!]_\Delta(v) &\triangleq \exists w_1, w_2, w_1', w2'.\ v = (w_1, w_2) \land v' = (w_1', w_2') \land\\
&\qquad [\![\Xi \vdash \tau]\!]_\Delta(w_1, w_1') * [\![\Xi \vdash \tau']\!]_\Delta(w_2, w_2')\\
[\![\Xi \vdash \mu\alpha.\ \tau]\!]_\Delta(v, v') &\triangleq \mu f : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}.\ \exists w, w'.\ v = \mathsf{fold}\, w \land v' = \mathsf{fold}\, w' \land\\
&\qquad \triangleright [\![\alpha, \Xi \vdash \tau]\!]_{\Delta, \alpha \mapsto f}(w, w')\\
[\![\Xi \vdash \tau \to \tau']\!]_\Delta(v, v') &\triangleq \square \left(\forall w, w'.\ [\![\Xi \vdash \tau]\!]_\Delta(w, w') \Rightarrow \mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(v\ w, v'\ w')\right)\\
[\![\Xi \vdash \forall\alpha.\ \tau]\!]_\Delta(v, v') &\triangleq \forall f : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}.\\
&\qquad \mathsf{persistent}(f) \Rightarrow \square \left(\mathcal{E}[\![\alpha, \Xi \vdash \tau]\!]_{\Delta, \alpha \mapsto f}(v\ \_, v'\ \_)\right)\\
[\![\Xi \vdash \mathsf{ref}(\tau)]\!]_\Delta(v, v') &\triangleq \exists \ell, \ell'.\ v = \ell \land v' = \ell' \land\\
&\qquad \boxed{\exists w, w'.\ \ell \mapsto_i w * \ell' \mapsto_s w' * [\![\Xi \vdash \tau]\!]_\Delta(w, w')}^{\mathcal{N}.\ell.\ell'}\\
[\![\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')
\end{aligned}$$

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

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

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

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

Logical relatedness ($\Xi \mid \Gamma \vDash e \leq_{\log} e' : \tau : \mathit{iProp}$): assuming $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$

$$\Xi \mid \Gamma \vDash e \leq_{\log} e' : \tau \triangleq \forall \Delta, \vec{v}, \vec{v'}.\ \left(\mathop{\Huge *}_{x_i : \tau_i} [\![\Xi \vdash \tau_i]\!]_\Delta(v_i, v_i')\right) \Rightarrow \mathcal{E}[\![\Xi \vdash \tau]\!]_\Delta(e[\vec{v}/\vec{x}], e'[\vec{v'}/\vec{x}])$$

8 Inadmissibility of the Bind Rule

Consider the derivation given the following derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{false} = \mathsf{false}}{\mathsf{wp}\ \mathsf{false}\ \{v.\ v = \mathsf{false}\}}}{\mathsf{wp}\ \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}\ \{v.\ v = \mathsf{false}\}}}{\mathsf{wp}\ \mathsf{true}\ \{w.\ \mathsf{wp}\ \mathsf{if}\ w\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}\ \{v.\ v = \mathsf{false}\}\}}}{\mathsf{wp}\ \mathsf{throw}\ \mathsf{true}\ \mathsf{to}\ -\ \{w.\ \mathsf{wp}\ \mathsf{if}\ w\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}\ \{v.\ v = \mathsf{false}\}\}}\ \text{INADMISSIBLE-BIND}}{\mathsf{wp}\ \mathsf{if}\ (\mathsf{throw}\ \mathsf{true}\ \mathsf{to}\ -)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true}\ \{v.\ v = \mathsf{false}\}}}{\mathsf{wp}\ \mathsf{call/cc}\ (x.\ \mathsf{if}\ (\mathsf{throw}\ \mathsf{true}\ \mathsf{to}\ x)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true})\ \{v.\ v = \mathsf{false}\}}$$

Here, we omit steps corresponding to eliminations of $\triangleright$. Note that we can easily show that

$$\mathsf{call/cc}\ (x.\ \mathsf{if}\ (\mathsf{throw}\ \mathsf{true}\ \mathsf{to}\ x)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \mathsf{true})$$

reduces to the value true which falsifies the derivation above.

9 Two servers: the code

We write programs using OCaml like syntax. The command **abort** is a shorthand for throw () to cont($-$).

The code for table and its underlying spin lock.

```
let newLock () = ref false

let rec acquire l =
  if CAS(l, false, true) then () else acquire l

let release l = l := false

let rec list_get (n : ℕ) (l : α list) : α option =
  match l with
    nil -> None
  | h :: tl -> if (n = 0) then h else get (n - 1) tl

let get ((lc, tb) : lock ∗ (α list) ref) (n : ℕ) : α option =
  acquire lc; list_get n !tb; release ()

let rec snoc (l : α list) (x : α) : α list
  match l with
    nil -> x :: nil
  | h :: tl -> h :: (snoc tl x)

let rec length (l : α list) : ℕ
  match l with
    nil -> 0
  | h :: tl -> 1 (length tl)

let list_associate (l : (α list) ref) (x : α) : ℕ =
  let r = length l in l := snoc (!l) x; r

let associate ((lc, tb) : lock ∗ (α list) ref) (x : α) : ℕ =
  acquire lc; let i = list_associate tb x in release lc; i

let newTable () = (newLock (), ref [])
```

The code of the server and the handlers is as follows.

```
type contid = ℕ

type data = ℕ

type ServerConnT = (1 -> (option contid) × data) × ((contid + data) -> 1)

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

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 (inr (sum + n));
          writer (inl (associate tb (sum + n)));
          abort
      end
    | (None, n) ->
      writer (inr n);
      let cid = associate tb n
      in writer (inl cid)

let read_client tb writer =
  callcc (k, writer (inl (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
    | inr (None, n) ->
      let rec loop m =
        writer (inr m);
        let (v, reader, writer) = read_client () in
        loop (m + v) reader writer
      in
      loop n reader writer
```

A candidate client can be written as follows:

```
let handler = ... (* pick handler1 or handler2 *)

let make_chan : (1 -> α) × (α -> 1) =
 let l = ref None in
 let rec waitfor () =
  match !l with
    None -> waitfor ()
  | Some v => l := None; v
 in
 let rec write v =
  if CAS(l, None, Some v) then () else write v
 in
 (waitfor, write)

let newConnection () = (make_chan, make_chan)

let contact_sender =
  let (listener, sender) = make_chan in
  serve listener handler; sender

let send_receive cid number =
  let (reader1, writer1, reader2, writer2) = newConnection () in
  contact_server (reader2, writer1);
  writer1 (cid, number);
  reader2 ()

let client () =
  let (cid1, ans1) =
    send_receive None 10 in (* ans1 = 10 *)
  let (cid2, ans2) =
    send_receive (Some cid1) 5 in (* ans2 = 15 *)
  let (cid3, ans3) =
    send_receive (None) 17 in (* ans3 = 17 *)
  let (cid4, ans4) =
    send_receive (Some cid3) 9 in (* ans4 = 26 *)
  let (cid5, ans5) =
    send_receive (Some cid1) 19 in (* ans5 = 29 *)
  let (cid6, ans6) =
    send_receive (Some cid2) 17 in (* ans6 = 32 *)
  ()

let () = client ()
```

It starts a server with some handler and communicates with it.

10 One-shot 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 Støvring and Lassen [2007], Dreyer et al. [2012], with pen-and-paper proofs, not mechanized formal verication. Here we show that the equivalence also holds in our concurrent language (subtly so; because the 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:

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

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 $\mathfrak{CC}'$:

$$\mathfrak{CC}' \triangleq \Lambda\, \lambda f.\ \mathsf{let}\, \ell = \mathsf{ref}(\mathsf{cont}(-))\ \mathsf{in}\ G\ f$$

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

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 5.

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

We only discuss one side of the refinement, namely, $\mathfrak{CC}' \leq_{\text{ctx}} \mathfrak{CC}$. 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:

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

Here K is the continuation that is captured by $\mathfrak{CC}$ and $\mathit{restore}(\ell)$ is the piece of code that takes a new one-shot continuations and restores ℓ:

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

Intuitively, it states that the continuation stored in ℓ is a one-shot continuation. Furthermore, we know that the continuation has never been used (as the one-shot bit b stores false).

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:

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

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

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. 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, $\mathfrak{CC} \leq_{\text{ctx}} \mathfrak{CC}'$ is simpler and follows basically 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.

10.1 The resource for one-shot bit

Now we discuss using Iris base logic's ghost state to model the one-shot bit resources: $\mathit{OneShotBits}(M)$ and $\mathit{isOneShotBit}(b)$.

$$\begin{aligned}
\textsc{OneShotBit} &\triangleq \textsc{Auth}(\mathsf{finset}(\ell)) \\
\mathit{OneShotBits}(M) &\triangleq \boxed{\bullet\, M}^{\gamma_{os}} \\
\mathit{isOneShotBit}(b) &\triangleq \boxed{\circ\, \{b\}}^{\gamma_{os}}
\end{aligned}$$

It is easy to see, based on the rules for monoids above, that $\mathit{isOneShotBit}(b)$ is persistent and that:

$$\mathit{OneShotBits}(M) * \mathit{isOneShotBit}(b) \vdash \checkmark(\bullet\, M \cdot \circ\, \{b\}) \vdash \{b\} \subseteq M \vdash b \in M$$

References

A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.

A. Appel, P.-A. Melliès, C. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.

D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *LMCS*, 7 (2:16), 2011.

D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012. doi: 10.1017/S095679681200024X.

D. P. Friedman and C. T. Haynes. Constraining control. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 245–254, New York, NY, USA, 1985. ACM. ISBN 0-89791-147-4. doi: 10.1145/318593.318654. URL `http://doi.acm.org/10.1145/318593.318654`.

R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650, 2015.

R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-order ghost state. In *ICFP*, pages 256–269, 2016.

R. Krebbers, R. Jung, A. Bizjak, J.-H. Jourdan, D. Dreyer, and L. Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming (ESOP)*, April 2017a.

R. Krebbers, A. Timany, and L. Birkedal. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*, 2017b.

K. Støvring and S. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *POPL*, 2007.