Robustly-Safe Compilation – Technical Appendix

Marco Patrignani[1,2] Deepak Garg[3]
[1] Stanford University
[2] CISPA Helmholz Center for Information Security
[3] Max Planck Institute for Software Systems

Contents

1 The Untyped Source Language: L^{U}

This is a sequential while language with monitors.

1.1 Syntax

$$\begin{aligned}
\textit{Whole Programs}\ \mathsf{P} &::= \ell_{\mathsf{root}}; \mathsf{H}; \overline{\mathsf{F}}; \overline{\mathsf{I}}\\
\textit{Components}\ \mathsf{C} &::= \ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}\\
\textit{Contexts}\ \mathsf{A} &::= \mathsf{H}; \overline{\mathsf{F}}\,[\cdot]\\
\textit{Interfaces}\ \mathsf{I} &::= \mathsf{f}\\
\textit{Functions}\ \mathsf{F} &::= \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return};\\
\textit{Operations}\ \oplus &::= + \mid -\\
\textit{Comparison}\ \otimes &::= == \mid < \mid >\\
\textit{Values}\ \mathsf{v} &::= \mathsf{b} \in \{\mathsf{true}, \mathsf{false}\} \mid \mathsf{n} \in \mathbb{N} \mid \langle \mathsf{v}, \mathsf{v} \rangle \mid \ell\\
\textit{Expressions}\ \mathsf{e} &::= \mathsf{x} \mid \mathsf{v} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e} \otimes \mathsf{e} \mid \langle \mathsf{e}, \mathsf{e} \rangle \mid \mathsf{e.1} \mid \mathsf{e.2} \mid \mathsf{!e}\\
\textit{Statements}\ \mathsf{s} &::= \mathsf{skip} \mid \mathsf{s}; \mathsf{s} \mid \mathsf{let\ x = e\ in\ s} \mid \mathsf{if\ e\ then\ s\ else\ s}\\
&\quad \mid \mathsf{call\ f\ e} \mid \mathsf{let\ x = new\ e\ in\ s} \mid \mathsf{x := e}\\
\textit{Eval. Ctxs.}\ \mathsf{E} &::= [\cdot] \mid \mathsf{e} \oplus \mathsf{E} \mid \mathsf{E} \oplus \mathsf{n} \mid \mathsf{e} \otimes \mathsf{E} \mid \mathsf{E} \otimes \mathsf{n}\\
&\quad \mid \langle \mathsf{e}, \mathsf{E} \rangle \mid \langle \mathsf{E}, \mathsf{v} \rangle \mid \mathsf{E.1} \mid \mathsf{E.2} \mid \mathsf{!E}\\
\textit{Heaps}\ \mathsf{H} &::= \varnothing \mid \mathsf{H}; \ell \mapsto \mathsf{v}\\
\textit{Monitors}\ \mathsf{M} &::= (\{\sigma\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}})\\
\textit{Mon. States}\ \sigma &\in \mathcal{S}\\
\textit{Mon. Reds.}\ \rightsquigarrow &::= \varnothing \mid \rightsquigarrow; (\mathsf{s}, \mathsf{H}, \mathsf{s})\\
\textit{Substitutions}\ \rho &::= \varnothing \mid \rho[\mathsf{v} \ / \ \mathsf{x}]\\
\textit{Prog. States}\ \Omega &::= \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}}\\
\textit{Labels}\ \lambda &::= \epsilon \mid \alpha\\
\textit{Actions}\ \alpha &::= \mathtt{call}\ \mathsf{f}\ \mathsf{v}\ \mathsf{H}? \mid \mathtt{call}\ \mathsf{f}\ \mathsf{v}\ \mathsf{H}! \mid \mathtt{ret}\ \mathsf{H}! \mid \mathtt{ret}\ \mathsf{H}?\\
\textit{Traces}\ \overline{\alpha} &::= \varnothing \mid \overline{\alpha} \cdot \alpha
\end{aligned}$$

1.2 Dynamic Semantics

Rules L^{U}-Jump-Internal to L^{U}-Jump-OUT dictate the kind of a jump between two functions: if internal to the component/attacker, in(from the attacker to the component) or out(from the component to the attacker). Rule L^{U}-Plug tells how to obtain a whole program from a component and an attacker. Rule L^{U}-Whole tells when a program is whole. Rule L^{U}-Initial State tells the initial state of a whole program. Rule L^{U}-Monitor Step tells when a monitor makes a single step given a heap.

Helpers

$$\dfrac{\begin{array}{c}((f' \in \overline{I} \land f \in \overline{I}) \lor \\ (f' \notin \overline{I} \land f \notin \overline{I}))\end{array}}{\overline{I} \vdash f, f' : \mathsf{internal}}\;(L^{U}\text{-Jump-Internal}) \qquad \dfrac{f \in \overline{I} \land f' \notin \overline{I}}{\overline{I} \vdash f, f' : \mathsf{in}}\;(L^{U}\text{-Jump-IN}) \qquad \dfrac{f \notin \overline{I} \land f' \in \overline{I}}{\overline{I} \vdash f, f' : \mathsf{out}}\;(L^{U}\text{-Jump-OUT})$$

$$\dfrac{\begin{array}{c}A \equiv H; \overline{F}\,[\cdot] \qquad C \equiv \ell_{root}; \overline{F'}; \overline{I} \qquad \vdash C, \overline{F} : \mathsf{whole} \\ \mathsf{main} \in \mathtt{names}(\overline{F})\end{array}}{A\,[C] = \ell_{root}; H; \ell_{root} \mapsto 0; \overline{F; F'}; \overline{I}}\;(L^{U}\text{-Plug})$$

$$\dfrac{\begin{array}{cc}C \equiv \ell_{root}; \overline{F'}; \overline{I} & \mathtt{names}(\overline{F}) \cap \mathtt{names}(\overline{F'}) = \varnothing \\ \mathtt{names}(\overline{I}) \subseteq \mathtt{names}(\overline{F}) \cup \mathtt{names}(\overline{F'}) & \mathtt{fv}(\overline{F}) \cup \mathtt{fv}(\overline{F'}) = \varnothing\end{array}}{\vdash C, \overline{F} : \mathsf{whole}}\;(L^{U}\text{-Whole})$$

$$\dfrac{P \equiv \ell_{root}; H; \overline{F}; \overline{I} \qquad C \equiv \ell_{root}; \overline{F}; \overline{I} \qquad \mathsf{main}(x) \mapsto s; \mathsf{return}; \in \overline{F}}{\Omega_0\,(P) = C; H, \ell_{root} \mapsto 0 \triangleright (s[0 \,/\, x])_{\mathsf{main}}}\;(L^{U}\text{-Initial State})$$

1.2.1 Component Semantics

$H \triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow e'$ Expression e reduces to e'.

$C, H \triangleright s \xrightarrow{\epsilon} C', H' \triangleright s'$ Statement s reduces to s' and evolves the rest accordingly, emitting label λ.

$\Omega \xRightarrow{\overline{\alpha}} \Omega'$ Program state Ω steps to Ω' emitting trace $\overline{\alpha}$.

$$\boxed{H \triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow e'}$$

$$\dfrac{H \triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow e'}{H \triangleright E\,[e] \hookrightarrow\!\!\!\!\twoheadrightarrow E\,[e']}\;(EL^{U}\text{-ctx}) \qquad \dfrac{}{H \triangleright v \hookrightarrow\!\!\!\!\twoheadrightarrow v}\;(EL^{U}\text{-val}) \qquad \dfrac{}{H \triangleright \langle v, v' \rangle\,.1 \hookrightarrow\!\!\!\!\twoheadrightarrow v}\;(EL^{U}\text{-p1})$$

$$\dfrac{}{H \triangleright \langle v, v' \rangle\,.2 \hookrightarrow\!\!\!\!\twoheadrightarrow v'}\;(EL^{U}\text{-p2}) \qquad \dfrac{n \oplus n' = n''}{H \triangleright n \oplus n' \hookrightarrow\!\!\!\!\twoheadrightarrow n''}\;(EL^{U}\text{-op}) \qquad \dfrac{n \otimes n' = b}{H \triangleright n \otimes n' \hookrightarrow\!\!\!\!\twoheadrightarrow b}\;(EL^{U}\text{-comp})$$

$$\dfrac{\ell \mapsto v \in H}{H \triangleright !\ell \hookrightarrow\!\!\!\!\twoheadrightarrow v}\;(EL^{U}\text{-dereference})$$

$$\boxed{C, H \triangleright s \xrightarrow{\lambda} C', H' \triangleright s'}$$

$$\dfrac{}{C, H \triangleright \mathsf{skip}; s \xrightarrow{\epsilon} C, H \triangleright s}\;(EL^{U}\text{-sequence}) \qquad \dfrac{C, H \triangleright s \xrightarrow{\lambda} C, H' \triangleright s'}{C, H \triangleright s; s'' \xrightarrow{\lambda} C, H' \triangleright s'; s''}\;(EL^{U}\text{-step})$$

(EL^{U}-if-true)

$$\frac{H \triangleright e \hookrightarrow\!\!\!\!\rightarrow \text{true}}{C, H \triangleright \text{if } e \text{ then } s \text{ else } s' \xrightarrow{\epsilon} C, H \triangleright s}$$

(EL^{U}-if-false)

$$\frac{H \triangleright e \hookrightarrow\!\!\!\!\rightarrow \text{false}}{C, H \triangleright \text{if } e \text{ then } s \text{ else } s' \xrightarrow{\epsilon} C, H \triangleright s}$$

(EL^{U}-letin)

$$\frac{H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v}{C, H \triangleright \text{let } x = e \text{ in } s \xrightarrow{\epsilon} C, H \triangleright s[v \mathbin{/} x]}$$

(EL^{U}-alloc)

$$\frac{H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \qquad \ell \notin \mathtt{dom}(H)}{C, H \triangleright \text{let } x = \text{new } e \text{ in } s \xrightarrow{\epsilon} C, H; \ell \mapsto v \triangleright s[\ell \mathbin{/} x]}$$

(EL^{U}-update)

$$\frac{\begin{array}{c} H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \\ H = H_1; \ell \mapsto v'; H_2 \qquad H' = H_1; \ell \mapsto v; H_2 \end{array}}{C, H \triangleright \ell := e \xrightarrow{\epsilon} C, H' \triangleright \text{skip}}$$

(EL^{U}-call-internal)

$$\frac{\begin{array}{cc} \overline{C}.\mathtt{intfs} \vdash f, f' : \text{internal} & \overline{f'} = \overline{f''}; f' \\ f(x) \mapsto s; \text{return}; \in C.\mathtt{funs} & H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \end{array}}{C, H \triangleright (\text{call } f\ e)_{\overline{f'}} \xrightarrow{\epsilon} C, H \triangleright (s; \text{return}; [v \mathbin{/} x])_{\overline{f'}; f}}$$

(EL^{U}-callback)

$$\frac{\begin{array}{cc} \overline{f'} = \overline{f''}; f' & f(x) \mapsto s; \text{return}; \in \overline{F} \\ \overline{C}.\mathtt{intfs} \vdash f', f : \text{out} & H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \end{array}}{C, H \triangleright (\text{call } f\ e)_{\overline{f'}} \xrightarrow{\mathtt{call}\ f\ v\ H!} C, H \triangleright (s; \text{return}; [v \mathbin{/} x])_{\overline{f'}; f}}$$

(EL^{U}-call)

$$\frac{\begin{array}{cc} \overline{f'} = \overline{f''}; f' & f(x) \mapsto s; \text{return}; \in C.\mathtt{funs} \\ \overline{C}.\mathtt{intfs} \vdash f', f : \text{in} & H \triangleright e \hookrightarrow\!\!\!\!\rightarrow v \end{array}}{C, H \triangleright (\text{call } f\ e)_{\overline{f'}} \xrightarrow{\mathtt{call}\ f\ v\ H?} C, H \triangleright (s; \text{return}; [v \mathbin{/} x])_{\overline{f'}; f}}$$

(EL^{U}-ret-internal)

$$\frac{\overline{f'} = \overline{f''}; f' \qquad \overline{C}.\mathtt{intfs} \vdash f, f' : \text{internal}}{C, H \triangleright (\text{return};)_{\overline{f'}; f} \xrightarrow{\epsilon} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

(EL^{U}-retback)

$$\frac{\overline{f'} = \overline{f''}; f' \qquad \overline{C}.\mathtt{intfs} \vdash f, f' : \text{in}}{C, H \triangleright (\text{return};)_{\overline{f'}; f} \xrightarrow{\mathtt{ret}\ H?} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

(EL^{U}-return)

$$\frac{\overline{f'} = \overline{f''}; f' \qquad \overline{C}.\mathtt{intfs} \vdash f, f' : \text{out}}{C, H \triangleright (\text{return};)_{\overline{f'}; f} \xrightarrow{\mathtt{ret}\ H!} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

$$\boxed{\Omega \xRightarrow{\overline{\alpha}} \Omega'}$$

$$\text{(EL}^{\mathsf{U}}\text{-single)}\ \dfrac{\Omega \xrightarrow{\alpha} \Omega'}{\Omega \xRightarrow{\alpha} \Omega'} \qquad \text{(EL}^{\mathsf{U}}\text{-silent)}\ \dfrac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Longrightarrow \Omega'} \qquad \text{(EL}^{\mathsf{U}}\text{-trans)}\ \dfrac{\Omega \xRightarrow{\overline{\alpha}} \Omega'' \quad \Omega'' \xRightarrow{\overline{\alpha'}} \Omega'}{\Omega \xRightarrow{\overline{\alpha}\cdot\overline{\alpha'}} \Omega'}$$

1.3 Monitor Semantics

Let $\texttt{reach}(\ell_{\mathsf{o}}, \mathsf{H})$ return a set of locations $\{\ell\}$ in H such that it is possible to reach any $\ell \in \{\ell\}$ from ℓ_{o} just by expression evaluation.

$$\texttt{reach}(\ell, \mathsf{H}) = \{\ell \mid \exists \mathsf{e}.\ \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\!\rightarrow \ \ell \land \ell \in \texttt{dom}(\mathsf{H})\}$$

To ensure monitor transitions have a meaning, they are assumed to be closed under bijective renaming of locations.

$$\boxed{\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'}$$

$$\text{(L}^{\mathsf{U}}\text{-Monitor Step)}\ \dfrac{\begin{array}{cc} \mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}}) & \mathsf{M}' = (\{\sigma\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{f}}) \\ (\sigma_{\mathsf{c}}, \mathsf{H}', \sigma_{\mathsf{f}}) \in \rightsquigarrow \mathsf{H}' \subseteq \mathsf{H} & \texttt{dom}(\mathsf{H}') = \texttt{reach}(\ell_{\mathsf{root}}, \mathsf{H}) \end{array}}{\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'}$$

$$\text{(L}^{\mathsf{U}}\text{-Monitor Step Trace Base)}\ \dfrac{}{\mathsf{M}; \varnothing \rightsquigarrow \mathsf{M}} \qquad \text{(L}^{\mathsf{U}}\text{-Monitor Step Trace)}\ \dfrac{\mathsf{M}; \overline{\mathsf{H}} \rightsquigarrow \mathsf{M}'' \quad \mathsf{M}''; \mathsf{H} \rightsquigarrow \mathsf{M}'}{\mathsf{M}; \overline{\mathsf{H}} \cdot \mathsf{H} \rightsquigarrow \mathsf{M}'}$$

$$\text{(L}^{\mathsf{U}}\text{-valid trace)}\ \dfrac{\mathsf{M}; \overline{\mathsf{H}} \rightsquigarrow \mathsf{M}' \quad \texttt{heaps}(\overline{\alpha}) = \overline{\mathsf{H}}}{\mathsf{M} \vdash \overline{\alpha}}$$

Monitor actions are the only part of traces that matter for safety, so we define function $\texttt{heaps}(\,\cdot\,)$ that takes a general trace and elides all but the heap of actions. This function is used by both languages so we typeset it in black.

$$\begin{aligned} \texttt{heaps}(\varnothing) &= \varnothing \\ \texttt{heaps}(\texttt{call}\ f\ v\ H? \cdot \overline{\alpha}) &= H \cdot \texttt{heaps}(\overline{\alpha}) \\ \texttt{heaps}(\texttt{call}\ f\ v\ H! \cdot \overline{\alpha}) &= H \cdot \texttt{heaps}(\overline{\alpha}) \\ \texttt{heaps}(\texttt{ret}\ H! \cdot \overline{\alpha}) &= H \cdot \texttt{heaps}(\overline{\alpha}) \\ \texttt{heaps}(\texttt{ret}\ H? \cdot \overline{\alpha}) &= H \cdot \texttt{heaps}(\overline{\alpha}) \end{aligned}$$

2 The Target Language: $\mathbf{L^P}$

2.1 Syntax

$$
\begin{aligned}
\textit{Whole Programs}\ \mathbf{P} &::= \mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}\\
\textit{Components}\ \mathbf{C} &::= \mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}\\
\textit{Contexts}\ \mathbf{A} &::= \overline{\mathbf{F}}\,[\cdot]\\
\textit{Interfaces}\ \mathbf{I} &::= \mathbf{f}\\
\textit{Functions}\ \mathbf{F} &::= \mathbf{f(x)} \mapsto \mathbf{s}; \mathbf{return};\\
\textit{Operations}\ \oplus &::= + \mid -\\
\textit{Comparison}\ \otimes &::= == \mid < \mid >\\
\textit{Values}\ \mathbf{v} &::= \mathbf{n} \in \mathbb{N} \mid \langle \mathbf{v}, \mathbf{v} \rangle \mid \mathbf{k}\\
\textit{Expressions}\ \mathbf{e} &::= \mathbf{x} \mid \mathbf{v} \mid \mathbf{e} \oplus \mathbf{e} \mid \mathbf{e} \otimes \mathbf{e} \mid \langle \mathbf{e}, \mathbf{e} \rangle \mid \mathbf{e.1} \mid \mathbf{e.2} \mid \mathbf{!e\ with\ e}\\
\textit{Statements}\ \mathbf{s} &::= \mathbf{skip} \mid \mathbf{s;s} \mid \mathbf{let\ x = e\ in\ s} \mid \mathbf{ifz\ e\ then\ s\ else\ s} \mid \mathbf{call\ f\ e}\\
&\mid \mathbf{x := e\ with\ e} \mid \mathbf{let\ x = new\ e\ in\ s} \mid \mathbf{let\ x = hide\ e\ in\ s}\\
\textit{Eval. Ctxs.}\ \mathbf{E} &::= [\cdot] \mid \mathbf{e} \oplus \mathbf{E} \mid \mathbf{E} \oplus \mathbf{n} \mid \mathbf{e} \otimes \mathbf{E} \mid \mathbf{E} \otimes \mathbf{n} \mid \mathbf{!E\ with\ v} \mid \mathbf{!e\ with\ E}\\
&\mid \langle \mathbf{e}, \mathbf{E} \rangle \mid \langle \mathbf{E}, \mathbf{v} \rangle \mid \mathbf{E.1} \mid \mathbf{E.2}\\
\textit{Heaps}\ \mathbf{H} &::= \varnothing \mid \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \eta \mid \mathbf{H}; \mathbf{k}\\
\textit{Tag}\ \eta &::= \perp \mid \mathbf{k}\\
\textit{Monitors}\ \mathbf{M} &::= (\{\sigma\}, \rightsquigarrow, \sigma_0, \mathbf{k_{root}}, \sigma_c)\\
\textit{Mon. States}\ \sigma &\in \mathcal{S}\\
\textit{Mon. Reds.}\ \rightsquigarrow &::= \varnothing \mid \rightsquigarrow; (\mathbf{s}, \mathbf{H}, \mathbf{s})\\
\textit{Substitutions}\ \rho &::= \varnothing \mid \rho[\mathbf{v} \,/\, \mathbf{x}]\\
\textit{Prog. States}\ \mathbf{\Omega} &::= \mathbf{C}, \mathbf{H} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}\\
\textit{Labels}\ \lambda &::= \epsilon \mid \alpha\\
\textit{Actions}\ \alpha &::= \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}? \mid \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}! \mid \texttt{ret}\ \mathbf{H}! \mid \texttt{ret}\ \mathbf{H}?\\
\textit{Traces}\ \overline{\alpha} &::= \varnothing \mid \overline{\alpha} \cdot \alpha
\end{aligned}
$$

2.2 Operational Semantics of $\mathbf{L^P}$

Helpers

($\mathbf{L^P}$-Jump-Internal)

$$\frac{\begin{array}{c}((\mathbf{f'} \in \overline{\mathbf{I}} \land \mathbf{f} \in \overline{\mathbf{I}}) \lor \\ (\mathbf{f'} \notin \overline{\mathbf{I}} \land \mathbf{f} \notin \overline{\mathbf{I}}))\end{array}}{\overline{\mathbf{I}} \vdash \mathbf{f}, \mathbf{f'} : \mathbf{internal}}$$

($\mathbf{L^P}$-Jump-IN)

$$\frac{\mathbf{f} \in \overline{\mathbf{I}} \land \mathbf{f'} \notin \overline{\mathbf{I}}}{\overline{\mathbf{I}} \vdash \mathbf{f}, \mathbf{f'} : \mathbf{in}}$$

($\mathbf{L^P}$-Jump-OUT)

$$\frac{\mathbf{f} \notin \overline{\mathbf{I}} \land \mathbf{f'} \in \overline{\mathbf{I}}}{\overline{\mathbf{I}} \vdash \mathbf{f}, \mathbf{f'} : \mathbf{out}}$$

($\mathbf{L}^{\mathbf{P}}$-Plug)

$$\dfrac{\begin{array}{cc}\mathbf{A} \equiv \overline{\mathbf{F}}\,[\cdot] & \mathbf{C} \equiv \mathbf{k_{root}}\,; \overline{\mathbf{F'}}; \overline{\mathbf{I}} \\ \vdash \mathbf{C}, \overline{\mathbf{F}} : \mathbf{whole} & \mathbf{main(x) \mapsto s; return;} \in \overline{\mathbf{F}}\end{array}}{\mathbf{A}\,[\mathbf{C}] = \mathbf{k_{root}}\,; \overline{\mathbf{F}; \mathbf{F'}}; \overline{\mathbf{I}}}$$

($\mathbf{L}^{\mathbf{P}}$-Whole)

$$\dfrac{\begin{array}{c}\mathbf{C} \equiv \mathbf{k_{root}}\,; \overline{\mathbf{F'}}; \overline{\mathbf{I}} \\ \texttt{names}(\overline{\mathbf{F}}) \cap \texttt{names}(\overline{\mathbf{F'}}) = \varnothing \qquad \texttt{names}(\overline{\mathbf{I}}) \subseteq \texttt{names}(\overline{\mathbf{F}})\end{array}}{\vdash \mathbf{C}, \overline{\mathbf{F}} : \mathbf{whole}}$$

($\mathbf{L}^{\mathbf{P}}$-Initial State)

$$\dfrac{\mathbf{P} \equiv \mathbf{k_{root}}\,; \overline{\mathbf{F}}; \overline{\mathbf{I}} \qquad \mathbf{C} \equiv \mathbf{k_{root}}\,; \overline{\mathbf{F}}; \overline{\mathbf{I}} \qquad \mathbf{main(x) \mapsto s; return;} \in \overline{\mathbf{F}}}{\mathbf{\Omega_0}\,(\mathbf{P}) = \mathbf{C}, \mathbf{k_{root}}\,; \mathbf{0} \mapsto \mathbf{0} : \mathbf{k_{root}} \triangleright (\mathbf{s}[\mathbf{0}\ /\ \mathbf{x}])_{\mathbf{main}}}$$

2.2.1 Component Semantics

$\mathbf{H} \triangleright \mathbf{e} \ \hookrightarrow\!\!\!\!\to\ \mathbf{e'}$ Expression $\mathbf{e}$ reduces to $\mathbf{e'}$.

$\mathbf{C}, \mathbf{H} \triangleright \mathbf{s} \xrightarrow{\epsilon} \mathbf{C'}, \mathbf{H'} \triangleright \mathbf{s'}$ Statement $\mathbf{s}$ reduces to $\mathbf{s'}$ and evolves the rest accordingly, emitting label λ.

$\mathbf{\Omega} \overset{\overline{\alpha}}{\Longrightarrow} \mathbf{\Omega'}$ Program state $\mathbf{\Omega}$ steps to $\mathbf{\Omega'}$ emitting trace $\overline{\alpha}$.

$$\boxed{\mathbf{H} \triangleright \mathbf{e} \ \hookrightarrow\!\!\!\!\to\ \mathbf{e'}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-val)

$$\dfrac{}{\mathbf{H} \triangleright \mathbf{v} \ \hookrightarrow\!\!\!\!\to\ \mathbf{v}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-p1)

$$\dfrac{}{\mathbf{H} \triangleright \langle \mathbf{v}, \mathbf{v'} \rangle\,.\mathbf{1} \ \hookrightarrow\!\!\!\!\to\ \mathbf{v}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-p2)

$$\dfrac{}{\mathbf{H} \triangleright \langle \mathbf{v}, \mathbf{v'} \rangle\,.\mathbf{1} \ \hookrightarrow\!\!\!\!\to\ \mathbf{v'}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-op)

$$\dfrac{n \oplus n' = n''}{\mathbf{H} \triangleright \mathbf{n} \oplus \mathbf{n'} \ \hookrightarrow\!\!\!\!\to\ \mathbf{n''}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-comp)

$$\dfrac{\text{if } n \otimes n' = \text{true then } \mathbf{n''} = \mathbf{0} \text{ else } \mathbf{n''} = \mathbf{1}}{\mathbf{H} \triangleright \mathbf{n} \otimes \mathbf{n'} \ \hookrightarrow\!\!\!\!\to\ \mathbf{n''}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-deref-top)

$$\dfrac{\mathbf{n} \mapsto \mathbf{v} : \bot \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{!n\ with}\ \_ \ \hookrightarrow\!\!\!\!\to\ \mathbf{v}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-deref-k)

$$\dfrac{\mathbf{n} \mapsto (\mathbf{v}, \mathbf{k}) \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{!n\ with\ k} \ \hookrightarrow\!\!\!\!\to\ \mathbf{v}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-ctx)

$$\dfrac{\mathbf{H} \triangleright \mathbf{e} \ \hookrightarrow\!\!\!\!\to\ \mathbf{e'}}{\mathbf{H} \triangleright \mathbf{E}\,[\mathbf{e}] \ \hookrightarrow\!\!\!\!\to\ \mathbf{E}\,[\mathbf{e'}]}$$

$$\boxed{\mathbf{C}; \mathbf{H} \triangleright \mathbf{s} \xrightarrow{\lambda} \mathbf{C'}; \mathbf{H'} \triangleright \mathbf{s'}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-sequence)

$$\dfrac{}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{skip; s} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-step)

$$\dfrac{\mathbf{C}, \mathbf{H} \triangleright \mathbf{s} \xrightarrow{\lambda} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s'}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{s}; \mathbf{s''} \xrightarrow{\lambda} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s'}; \mathbf{s}}$$

($\mathbf{E}\mathbf{L}^{\mathbf{P}}$-if-true)

$$\dfrac{\mathbf{H} \triangleright \mathbf{e} \ \hookrightarrow\!\!\!\!\to\ \mathbf{0}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{ifz\ e\ then\ s\ else\ s'} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}}$$

(EL^P-if-false)

$$\frac{H \triangleright e \lhook\joinrel\twoheadrightarrow n \qquad n \not\equiv 0}{C, H \triangleright \text{ifz } e \text{ then } s \text{ else } s' \xrightarrow{\epsilon} C, H \triangleright s'}$$

(EL^P-letin)

$$\frac{H \triangleright e \lhook\joinrel\twoheadrightarrow v}{C, H \triangleright \text{let } x = e \text{ in } s \xrightarrow{\epsilon} C, H \triangleright s[v \;/\; x]}$$

(EL^P-new)

$$\frac{H = H_1; n \mapsto (v', \eta) \qquad H \triangleright e \lhook\joinrel\twoheadrightarrow v \qquad H' = H; n+1 \mapsto v : \bot}{C, H \triangleright \text{let } x = \text{new } e \text{ in } s \longrightarrow C, H' \triangleright s[n+1 \;/\; x]}$$

(EL^P-hide)

$$\frac{\begin{array}{cc} H \triangleright e \lhook\joinrel\twoheadrightarrow n & k \notin \texttt{dom}(H) \\ H = H_1; n \mapsto v : \bot; H_2 & H' = H_1; n \mapsto v : k; H_2; k \end{array}}{C, H \triangleright \text{let } x = \text{hide } e \text{ in } s \xrightarrow{\epsilon} C, H' \triangleright s[k \;/\; x]}$$

(EL^P-assign-top)

$$\frac{\begin{array}{c} H \triangleright e \lhook\joinrel\twoheadrightarrow v \\ H = H_1; n \mapsto \_ : \bot; H_2 \qquad H' = H_1; n \mapsto v : \bot; H_2 \end{array}}{C, H \triangleright n := e \text{ with } \_ \xrightarrow{\epsilon} C, H' \triangleright \text{skip}}$$

(EL^P-assign-k)

$$\frac{\begin{array}{cc} H \triangleright e \lhook\joinrel\twoheadrightarrow v & H \triangleright e' \lhook\joinrel\twoheadrightarrow k \\ H = H_1; n \mapsto \_ : k; H_2 & H' = H_1; n \mapsto v : k; H_2 \end{array}}{C, H \triangleright n := e \text{ with } e' \xrightarrow{\epsilon} C, H' \triangleright \text{skip}}$$

(EL^P-call-internal)

$$\frac{\begin{array}{cc} \overline{C}.\texttt{intfs} \vdash f, f' : \text{internal} & \overline{f'} = \overline{f''}; f' \\ f(x) \mapsto s; \text{return}; \in C.\texttt{funs} & H \triangleright e \lhook\joinrel\twoheadrightarrow v \end{array}}{C, H \triangleright (\text{call } f \ e)_{\overline{f'}} \xrightarrow{\epsilon} C, H \triangleright (s; \text{return};[v \;/\; x])_{\overline{f'};f}}$$

(EL^P-callback)

$$\frac{\begin{array}{c} \overline{f'} = \overline{f''}; f' \qquad f(x) \mapsto s; \text{return}; \in \overline{F} \\ \overline{C}.\texttt{intfs} \vdash f', f : \text{out} H \triangleright e \lhook\joinrel\twoheadrightarrow v \end{array}}{C, H \triangleright (\text{call } f \ e)_{\overline{f'}} \xrightarrow{\texttt{call f v H!}} C, H \triangleright (s; \text{return};[v \;/\; x])_{\overline{f'};f}}$$

(EL^P-call)

$$\frac{\begin{array}{cc} \overline{f'} = \overline{f''}; f' & f(x) \mapsto s; \text{return}; \in C.\texttt{funs} \\ \overline{C}.\texttt{intfs} \vdash f', f : \text{in} & H \triangleright e \lhook\joinrel\twoheadrightarrow v \end{array}}{C, H \triangleright (\text{call } f \ e)_{\overline{f'}} \xrightarrow{\texttt{call f v H?}} C, H \triangleright (s; \text{return};[v \;/\; x])_{\overline{f'};f}}$$

(EL^P-ret-internal)

$$\frac{\overline{C}.\texttt{intfs} \vdash f, f' : \text{internal} \qquad \overline{f'} = \overline{f''}; f'}{C, H \triangleright (\text{return};)_{\overline{f'};f} \xrightarrow{\epsilon} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

(EL^P-retback)

$$\frac{\overline{C}.\texttt{intfs} \vdash f, f' : \text{in} \qquad \overline{f'} = \overline{f''}; f'}{C, H \triangleright (\text{return};)_{\overline{f'};f} \xrightarrow{\texttt{ret H?}} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

$$\frac{\overline{\mathbf{C}}.\mathtt{intfs} \vdash \mathbf{f}, \mathbf{f}' : \mathbf{out} \qquad \overline{\mathbf{f}'} = \overline{\mathbf{f}''}; \mathbf{f}'}{\mathbf{C}, \mathbf{H} \triangleright (\mathbf{return};)_{\overline{\mathbf{f}'};\mathbf{f}} \xrightarrow{\mathtt{ret}\ \mathbf{H}!} \mathbf{C}, \mathbf{H} \triangleright (\mathbf{skip})_{\overline{\mathbf{f}'}}}\ (\mathsf{E}\mathbf{L}^{\mathbf{P}}\text{-return})$$

$$\boxed{\mathbf{\Omega} \overset{\overline{\alpha}}{\Longrightarrow} \mathbf{\Omega}'}$$

$$\frac{\mathbf{\Omega} \xrightarrow{\alpha} \mathbf{\Omega}'}{\mathbf{\Omega} \overset{\alpha}{\Longrightarrow} \mathbf{\Omega}'}\ (\mathsf{E}\mathbf{L}^{\mathbf{P}}\text{-single}) \qquad \frac{\mathbf{\Omega} \xrightarrow{\epsilon} \mathbf{\Omega}'}{\mathbf{\Omega} \Longrightarrow \mathbf{\Omega}'}\ (\mathsf{E}\mathbf{L}^{\mathbf{P}}\text{-silent}) \qquad \frac{\begin{array}{c}\mathbf{\Omega} \overset{\overline{\alpha}}{\Longrightarrow} \mathbf{\Omega}'' \\ \mathbf{\Omega}'' \overset{\overline{\alpha'}}{\Longrightarrow} \mathbf{\Omega}'\end{array}}{\mathbf{\Omega} \overset{\overline{\alpha}\cdot\overline{\alpha'}}{\Longrightarrow} \mathbf{\Omega}'}\ (\mathsf{E}\mathbf{L}^{\mathbf{P}}\text{-trans})$$

2.3 Monitor Semantics

Define $\mathtt{reach}(\mathbf{n_r}, \mathbf{k_r}, \mathbf{H})$ as the set of locations $\{\mathbf{n}\}$ such that it is possible to reach any $\mathbf{n} \in \{\mathbf{n}\}$ from $\mathbf{n_r}$ using any expression and relying on capability $\mathbf{k_r}$ as well as any capability reachable from $\mathbf{n_r}$. Formally:

$$\mathtt{reach}(\mathbf{n_r}, \mathbf{k_r}, \mathbf{H}) = \left\{ \mathbf{n} \ \middle|\ \begin{array}{l} \mathbf{H} \triangleright \mathbf{e} \ \hookrightarrow\!\!\!\!\twoheadrightarrow\ \mathbf{!n\ with\ v} \ \hookrightarrow\!\!\!\!\twoheadrightarrow\ \mathbf{v}' \\ \mathtt{fv}(\mathbf{e}) = \mathbf{n_r} \cup \mathbf{k_r} \end{array} \right\}$$

$$\boxed{\mathbf{M}; \mathbf{H} \rightsquigarrow \mathbf{M}'}$$

$$\frac{\begin{array}{cc}\mathbf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma_{\mathbf{c}}) & \mathbf{M}' = (\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma_{\mathbf{f}}) \\ (\mathbf{s_c}, \mathbf{H}', \mathbf{s_f}) \in \rightsquigarrow \mathbf{H}' \subseteq \mathbf{H} & \mathtt{dom}(\mathbf{H}') = \mathtt{reach}(\mathbf{0}, \mathbf{k_{root}}, \mathbf{H})\end{array}}{\mathbf{M}; \mathbf{H} \rightsquigarrow \mathbf{M}'}\ (\mathbf{L}^{\mathbf{P}}\text{-Monitor Step})$$

$$\frac{}{\mathbf{M}; \varnothing \rightsquigarrow \mathbf{M}}\ (\mathbf{L}^{\mathbf{P}}\text{-Monitor Step Trace Base}) \qquad \frac{\mathbf{M}; \overline{\mathbf{H}} \rightsquigarrow \mathbf{M}'' \qquad \mathbf{M}''; \mathbf{H} \rightsquigarrow \mathbf{M}'}{\mathbf{M}; \overline{\mathbf{H}} \cdot \mathbf{H} \rightsquigarrow \mathbf{M}'}\ (\mathbf{L}^{\mathbf{P}}\text{-Monitor Step Trace})$$

$$\frac{\mathbf{M}; \overline{\mathbf{H}} \rightsquigarrow \mathbf{M}' \qquad \mathtt{heaps}(\overline{\alpha}) = \overline{\mathbf{H}}}{\mathbf{M} \vdash \overline{\alpha}}\ (\mathbf{L}^{\mathbf{P}}\text{-valid trace})$$

3 Language and Compiler Properties

3.1 Safety, Attackers and Robust Safety

These properties hold for both languages are written in black and only once.

Definition 1 (Safety)**.**

$$
\begin{array}{c}
M \vdash C : \mathit{safe} \overset{\mathsf{def}}{=} . \\
\text{if } \vdash C : \mathit{whole} \\
\text{then} \quad \text{if } \Omega_0\left(C\right) \overset{\overline{\alpha}}{\Longrightarrow} \_ \\
\text{then } M \vdash \overline{\alpha}
\end{array}
$$

A program is safe for a monitor if the monitor accepts any trace the program generates.

For now, we give an informal definition for function $\texttt{locs}(A)$, which returns all the locations statically bound in the heap and code of attacker A.

Definition 2 ((Informal) Attacker)**.**

$$C \vdash A : \mathit{attacker} \overset{\mathsf{def}}{=} \text{ no location the component cares about } \in \texttt{locs}(A)$$

An attacker is valid if it does not refer to the locations the component cares about. We leave the notion of *location the component cares about* abstract and instantiate it on a per-language basis later on.

Definition 3 (Robust Safety)**.**

$$
\begin{array}{c}
M \vdash C : \mathit{rs} \overset{\mathsf{def}}{=} \forall A. \\
\text{if } M \frown C \\
C \vdash A : \mathit{attacker} \\
\text{then } M \vdash A\left[C\right] : \mathit{safe}
\end{array}
$$

A program is robustly safe if it is safe for any attacker it is composed with.

The definition of $M \frown C$ is to be specified on a language-specific basis, as the next section does for L^{U} and $\mathbf{L}^{\mathbf{P}}$.

3.2 Monitor Agreement and Attacker for $\mathbf{L}^{\mathbf{P}}$ and L^{U}

Definition 4 (L^{U}: $\mathsf{M} \frown \mathsf{C}$)**.**

$$\left(\left\{\sigma\right\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}}\right) \frown \left(\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}\right)$$

A monitor and a component agree if they focus on the same initial location ℓ_{root}.

Definition 5 ($\mathbf{L^P}$: $\mathbf{M} \frown \mathbf{C}$)**.**

$$(\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{k_{root}}, \sigma_{\mathbf{c}}) \frown (\mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}})$$

A monitor and a component agree if they use the same capabilty $\mathbf{k_{root}}$ to protect the initial location $\mathbf{0}$.

To define attackers, we define functions $\texttt{locs}(\mathsf{A})$, which returns all the locations bound in the code and heap of A and function $\texttt{caps}(\mathbf{A})$, which returns all the capabilities bound in the code of $\mathbf{A}$. Intuitively, the former is defined inductively on the structure of functions, then statements, then expressions, where it collects all ℓ in an expression e, and also on the structure of heaps, where it collects all ℓ in a binding of the form $\ell' \mapsto \ell$. The latter is defined inductively on the structure of functions, then statements, then expressions, where it collects all $\mathbf{k}$ in an expression $\mathbf{e}$.

Definition 6 ($\mathsf{L^U}$ attacker)**.**

$$\mathsf{C} \vdash \mathsf{A} : \mathsf{attacker} \overset{\mathsf{def}}{=} \mathsf{C} = (\ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}), \mathsf{A} = \mathsf{H}; \overline{\mathsf{F'}}$$
$$\ell_{\mathsf{root}} \notin \texttt{locs}(\mathsf{A})$$

Definition 7 ($\mathbf{L^P}$ attacker)**.**

$$\mathbf{C} \vdash \mathbf{A} : \mathbf{attacker} \overset{\mathsf{def}}{=} \mathbf{C} = (\mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}}), \mathbf{A} = \overline{\mathbf{F'}}$$
$$\mathbf{k_{root}} \notin \texttt{caps}(\overline{\mathbf{F'}})$$

3.3 Cross-language Relations

Assume a partial bijection $\beta : \ell \times \mathbf{n} \times \eta$ from source to target heap locations such that

- if $(\ell_1, \mathbf{n}, \eta) \in \beta$ and $(\ell_2, \mathbf{n}, \eta)$ then $\ell_1 = \ell_2$;
- if $(\ell, \mathbf{n_1}, \eta_1) \in \beta$ and $(\ell, \mathbf{n_2}, \eta_2)$ then $\mathbf{n_1} = \mathbf{n_2}$ and $\eta_1 = \eta_2$.

we use this bijection to parametrise the relation so that we can relate meaningful locations.

For compiler correctness we rely on a β_0 which relates initial locations of monitors.

Assume a relation $\approx_\beta : \mathsf{v} \times \beta \times \mathbf{v}$ that is total so it maps any source value to a target value $\mathbf{v}$.

- $\forall \mathsf{v}. \exists \mathbf{v}. \mathsf{v} \approx_\beta \mathbf{v}$.

This relation is used for defining compiler correctness. By inspecting the semantics of $\mathsf{L^U}$, Rules $\mathrm{EL^P}$-sequence and $\mathrm{EL^P}$-if-true let us derive that

- $\mathsf{true} \approx_\beta \mathbf{0}$;
- $\mathsf{false} \approx_\beta \mathbf{n}$ where $\mathbf{n} \neq \mathbf{0}$;

- $\ell \approx_\beta \langle \mathbf{n}, \mathbf{v} \rangle$ where $\begin{cases} \mathbf{v} = \mathbf{k} & \text{if } (\ell, \mathbf{n}, \mathbf{k}) \in \beta \\ \mathbf{v} \neq \mathbf{k} & \text{otherwise, so } (\ell, \mathbf{n}, \bot) \in \beta \end{cases}$
- $\langle \mathsf{v}_1, \mathsf{v}_2 \rangle \approx_\beta \langle \mathbf{v_1}, \mathbf{v_2} \rangle$ iff $\mathsf{v}_1 \approx_\beta \mathbf{v_1}$ and $\mathsf{v}_2 \approx_\beta \mathbf{v_2}$.

We overload the notation and use the same notation to indicate the (assumed) relation between monitor states: $\sigma \approx \sigma$.

We lift this relation to sets of states point-wise and indicate it as follows: $\{\sigma\} \approx \{\sigma\}$. In these cases the bijection β is not needed as states do not have locations inside.

Function names are related when they are the same: $\mathsf{f} \approx_\beta \mathbf{f}$.

Variables names are related when they are the same: $\mathsf{x} \approx_\beta \mathbf{x}$.

Substitutions are related when they replace related values for related variables: $[\mathsf{v} \,/\, \mathsf{x}] \approx_\beta [\mathbf{v} \,/\, \mathbf{x}]$ iff $\mathsf{v} \approx_\beta \mathbf{v}$ and $\mathsf{x} \approx_\beta \mathbf{x}$.

$$\boxed{\alpha \approx_\beta \alpha}$$

$$\text{(Call relation)}\ \frac{\mathsf{f} \approx \mathbf{f} \qquad \mathsf{v} \approx_\beta \mathbf{v} \qquad \mathsf{H} \approx_\beta \mathbf{H}}{\mathtt{call}\ \mathsf{f}\ \mathsf{v}\ \mathsf{H}? \approx_\beta \mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?} \qquad \text{(Callback relation)}\ \frac{\mathsf{f} \approx \mathbf{f} \qquad \mathsf{v} \approx_\beta \mathbf{v} \qquad \mathsf{H} \approx_\beta \mathbf{H}}{\mathtt{call}\ \mathsf{f}\ \mathsf{v}\ \mathsf{H}! \approx_\beta \mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!}$$

$$\text{(Return relation)}\ \frac{\mathsf{H} \approx_\beta \mathbf{H}}{\mathtt{ret}\ \mathsf{H}! \approx_\beta \mathtt{ret}\ \mathbf{H}!} \qquad \text{(Returnback relation)}\ \frac{\mathsf{H} \approx_\beta \mathbf{H}}{\mathtt{ret}\ \mathsf{H}? \approx_\beta \mathtt{ret}\ \mathbf{H}?} \qquad \text{(Epsilon relation)}\ \frac{}{\epsilon \approx_\beta \epsilon}$$

Definition 8 ($\mathsf{M}\mathcal{R}\mathbf{M}$)**.** Given a monitor-specific relation $\sigma \approx \sigma$ on monitor states, we say that a relation $\mathcal{R}$ on source and target monitors is a *bisimulation* if the following hold whenever $\mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma_{\mathsf{c}})$ and $\mathbf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_\mathbf{0}, \mathbf{k_{root}}, \sigma_\mathbf{c})$ are related by $\mathcal{R}$:

1. $\sigma_0 \approx \sigma_\mathbf{0}$, and
2. $\sigma_\mathsf{c} \approx \sigma_\mathbf{c}$, and
3. For all β containing $(\ell_{\mathsf{root}}, \mathbf{0}, \mathbf{k_{root}})$ and all $\mathsf{H}, \mathbf{H}$ with $\mathsf{H} \approx_\beta \mathbf{H}$ the following hold:
   (a) $(\sigma_\mathsf{c}, \mathsf{H}, \_) \in \rightsquigarrow$ iff $(\sigma_\mathbf{c}, \mathbf{H}, \_) \in \rightsquigarrow$, and
   (b) $(\sigma_\mathsf{c}, \mathsf{H}, \sigma') \in \rightsquigarrow$ and $(\sigma_\mathbf{c}, \mathbf{H}, \sigma') \in \rightsquigarrow$ imply $(\{\sigma\}, \rightsquigarrow, \sigma_0, \ell_{\mathsf{root}}, \sigma')\mathcal{R}(\{\sigma\}, \rightsquigarrow, \sigma_\mathbf{0}, \mathbf{k_{root}}, \sigma')$.

Definition 9 ($\mathsf{M} \approx \mathbf{M}$)**.** $\mathsf{M} \approx \mathbf{M}$ is the union of all bisimulations $\mathsf{M}\mathcal{R}\mathbf{M}$, which is also a bisimulation.

$$\boxed{\mathsf{H} \approx_\beta \mathbf{H}}$$

$$\text{(Heap relation)}\ \frac{\begin{array}{c}\mathsf{H} \approx_\beta \mathbf{H_1}; \mathbf{H_2} \qquad \ell \approx_\beta \langle \mathbf{n}, \eta \rangle \qquad \mathsf{v} \approx_\beta \mathbf{v} \\ \mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto \mathbf{v} : \eta; \mathbf{H_2}\end{array}}{\mathsf{H}; \ell \mapsto \mathsf{v} \approx_\beta \mathbf{H}} \qquad \text{(Empty relation)}\ \frac{}{\varnothing \approx_\beta \overline{\mathbf{k}}}$$

The heap relation is crucial. A source heap H is related to a target heap $\mathbf{H}$ if for any location pointing to a value in the former, a related location points to a related value in the target (Rule Heap relation). The base case (Rule Empty relation) considers that in the target heap we may have keys, which are not related to source elements.

As additional notation for states, we define when a state is stuck as follows

$$\text{(Stuck state)}\quad \frac{\mathit{\Omega} = M; \overline{F}; \overline{I}; H \triangleright s \qquad s \not\equiv \mathit{skip} \qquad \nexists \mathit{\Omega}', \lambda . \mathit{\Omega} \xrightarrow{\lambda}\!\!\!\!\rightarrow \mathit{\Omega}'}{\mathit{\Omega}^{\times}}$$

A state that terminated is defined as follows; this definition is given for a concurrent version of the language too (this is relevant for languages defined later):

$$\text{(Terminated state)}\quad \frac{\mathit{\Omega} = M; \overline{F}; \overline{I}; H \triangleright \mathit{skip}}{\mathit{\Omega}^{\Downarrow}}$$

$$\text{(Terminated soup)}\quad \frac{\mathit{\Omega} = M; \overline{F}; \overline{I}; H \triangleright \mathit{\Pi} \qquad \forall \pi \in \mathit{\Pi} .\, M; \overline{F}; \overline{I}; H \triangleright \pi^{\Downarrow}}{\mathit{\Omega}^{\Downarrow}}$$

To define compiler correctness, we rely on a cross-language relation for program states. Two states are related if their monitors are related and if their whole heap is related (Rule Related states – Whole).

$$\boxed{\mathsf{\Omega} \approx_{\beta} \mathbf{\Omega}}$$

$$\text{(Related states – Whole)}\quad \frac{\begin{array}{c} \mathsf{\Omega} = \mathsf{M}; \overline{\mathsf{F}}, \overline{\mathsf{F'}}; \overline{\mathsf{I}}; \mathsf{H} \triangleright \mathsf{s} \\ \mathbf{\Omega} = \mathbf{M}; \overline{\mathbf{F}}, [\![\overline{\mathsf{F'}}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}; \overline{\mathbf{I}}; \mathbf{H} \triangleright \mathbf{s} \\ \mathsf{M} \approx_{\beta} \mathbf{M} \qquad\qquad \mathsf{H} \approx_{\beta} \mathbf{H} \end{array}}{\mathsf{\Omega} \approx_{\beta} \mathbf{\Omega}}$$

3.4 Correct and Robustly-safe Compilation

Consider a compiler to be a function of this form: $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathsf{C} \to \mathbf{C}$, taking a source component and producing a target component.

Definition 10 (Correct Compilation)**.**

$$\begin{aligned} \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : CC \overset{\mathsf{def}}{=} \forall \mathsf{C}, \exists \beta .& \\ \text{if } & \mathbf{\Omega_0}\left([\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right) \xRightarrow{\overline{\alpha}} \mathbf{\Omega} \\ & \mathbf{\Omega}^{\Downarrow} \\ & \mathsf{\Omega_0}\left(\mathsf{C}\right) \approx_{\beta_0} \mathbf{\Omega_0}\left([\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right) \end{aligned}$$

$$\begin{aligned}
\text{then}\quad & \mathsf{\Omega_0}\left(\mathsf{C}\right) \xRightarrow{\overline{\mathsf{\alpha}}} \mathsf{\Omega}\\
& \beta_0 \subseteq \beta\\
& \mathsf{\Omega} \approx_{\beta} \mathbf{\Omega}\\
& \overline{\mathsf{\alpha}} \approx_{\beta} \overline{\boldsymbol{\alpha}}\\
& \mathsf{\Omega}^{\Downarrow}
\end{aligned}$$

Technically, any sequence $\overline{\alpha}$ above is empty, as $\overline{\mathsf{I}}$ is empty (the program is whole).

Definition 11 (Robust Safety Preserving Compilation)**.**

$$\begin{aligned}
\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{RSC} \overset{\mathsf{def}}{=} & \forall \mathsf{C}, \mathsf{M}, \mathbf{M}.\\
\text{if}\quad & \mathsf{M} \vdash \mathsf{C} : \mathsf{rs}\\
& \mathsf{M} \approx \mathbf{M}\\
\text{then}\quad & \mathbf{M} \vdash [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathbf{rs}
\end{aligned}$$

3.4.1 Alternative definition for RSC

Definition 12 (Property-Free RSC)**.**

$$\begin{aligned}
\vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{PF\text{-}RSC} \overset{\mathsf{def}}{=} & \forall \mathsf{C}.\\
\text{if}\quad & \forall \mathbf{A}, \overline{\boldsymbol{\alpha}}.\\
& [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} \vdash \mathbf{A} : \mathbf{attacker}\\
& \vdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right] : \mathbf{whole}\\
& \mathbf{\Omega_0}\left([\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right) \xRightarrow{\overline{\boldsymbol{\alpha}}} \_\\
\text{then}\quad & \exists \mathsf{A}, \overline{\mathsf{\alpha}}.\\
& \mathsf{C} \vdash \mathsf{A} : \mathsf{attacker}\\
& \vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}\\
& \mathsf{\Omega_0}\left(\mathsf{C}\right) \xRightarrow{\overline{\mathsf{\alpha}}} \_\\
& \mathtt{heaps}(\overline{\mathsf{\alpha}}) \approx_{\beta} \mathtt{heaps}(\overline{\boldsymbol{\alpha}})
\end{aligned}$$

The property-free characterisation of RSC is equivalent to its original characterisation.

Theorem 1 (*PF-RSC* and *RSC* are equivalent)**.**

$$\forall [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}, \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{PF\text{-}RSC} \iff \vdash [\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathit{RSC}$$

3.4.2 Compiling Monitors

We can change the definition of compiler to also compile the monitor so we are not given a target monitor related to the source one, but the compiler gives us that monitor. Consider this compiler to have this type and this notation: $\llbracket \boxed{\boxed{\cdot}} \rrbracket^{\mathsf{S}}_{\mathbf{T}} : \mathsf{C} \to \mathbf{C}$.

Definition 13 (Robustly-safe Compilation with Monitors)**.**

$$\begin{aligned}
\vdash \llbracket \boxed{\boxed{\cdot}} \rrbracket^{\mathsf{S}}_{\mathbf{T}} : \texttt{rs-pres(M)} \overset{\mathsf{def}}{=} {} & \forall \mathsf{C}, \mathsf{M}. \\
& \text{if } \ \mathsf{M} \vdash \mathsf{C} : \mathsf{rs} \\
& \text{then } \ \llbracket \boxed{\boxed{\mathsf{M}}} \rrbracket^{\mathsf{S}}_{\mathbf{T}} \vdash \llbracket \boxed{\boxed{\mathsf{C}}} \rrbracket^{\mathsf{S}}_{\mathbf{T}} : \mathbf{rs}
\end{aligned}$$

4 Compiler from $\mathsf{L^U}$ to $\mathbf{L^P}$

Definition 14 (Compiler $\mathsf{L^U}$ to $\mathbf{L^P}$)**.** $\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} : \mathsf{C} \to \mathbf{C}$

$\llbracket \mathsf{C} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$ is defined as follows:

$$\llbracket \ell_{\mathsf{root}}; \overline{\mathsf{F}}; \overline{\mathsf{I}}, \mathbf{M} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{k_{root}};\ \llbracket \overline{\mathsf{F}} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}};\ \llbracket \overline{\mathsf{I}} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Comp}$$

$$\llbracket \mathsf{f(x)} \mapsto \mathsf{s; return;} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{f(x)} \mapsto \llbracket \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}; \mathbf{return;} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Function}$$

$$\llbracket \mathsf{f} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{f} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Interfaces}$$

Expressions

$$\llbracket \mathsf{true} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{0} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-True}$$

$$\llbracket \mathsf{false} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{1} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-False}$$

$$\llbracket \mathsf{n} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{n} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-nat}$$

$$\llbracket \mathsf{x} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{x} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Var}$$

$$\llbracket \ell \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \langle \mathbf{n}, \mathbf{v} \rangle \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Loc}$$

$$\llbracket \langle \mathsf{e_1}, \mathsf{e_2} \rangle \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \left\langle \llbracket \mathsf{e_1} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}, \llbracket \mathsf{e_2} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \right\rangle \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Pair}$$

$$\llbracket \mathsf{e.1} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.1} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-P1}$$

$$\llbracket \mathsf{e.2} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.2} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-P2}$$

$$\llbracket \mathsf{!e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{!}\llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.1}\ \mathbf{with}\ \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.2} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Deref}$$

$$\llbracket \mathsf{e} \oplus \mathsf{e'} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \oplus \llbracket \mathsf{e'} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-op}$$

$$\llbracket \mathsf{e} \otimes \mathsf{e'} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \otimes \llbracket \mathsf{e'} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-cmp}$$

Statements

$$\llbracket \mathsf{skip} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{skip} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Skip}$$

$$\llbracket \mathsf{s_u}; \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \llbracket \mathsf{s_u} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}; \llbracket \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Seq}$$

$$\llbracket \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{let}\ \mathbf{x} = \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{in}\ \llbracket \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Letin}$$

$$\llbracket \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s_t}\ \mathsf{else}\ \mathsf{s_e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{ifz}\ \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{then}\ \llbracket \mathsf{s_t} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{else}\ \llbracket \mathsf{s_e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-If}$$

$$\begin{aligned} \llbracket \mathsf{let}\ \mathsf{x} = \mathsf{new}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = {} & \mathbf{let}\ \mathbf{x_{loc}} = \mathbf{new}\ \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{in} \\ & \mathbf{let}\ \mathbf{x_{cap}} = \mathbf{hide}\ \mathbf{x_{loc}}\ \mathbf{in} \\ & \mathbf{let}\ \mathbf{x} = \langle \mathbf{x_{loc}}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ \llbracket \mathsf{s} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \end{aligned} \tag{$\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-New}$$

$$\begin{aligned}
\llbracket \mathsf{x} := \mathsf{e}' \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} &= \mathbf{let\ x1 = x.1\ in} && (\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\text{-Assign})\\
&\quad \mathbf{let\ x2 = x.2\ in}\\
&\quad\ \mathbf{x1} := \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{with\ x2}\\
\llbracket \mathsf{call\ f\ e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} &= \mathbf{call\ f}\ \llbracket \mathsf{e} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} && (\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\text{-call})
\end{aligned}$$

Note that the case for Rule ($\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-New) only works because we are in a sequential setting. In a concurrent setting an adversary could access $\mathbf{x_{loc}}$ before it is hidden, so the definition would change. See Rule ($\llbracket\cdot\rrbracket^{\mathsf{L^\tau}}_{\mathbf{L^\pi}}$-New) for a concurrent correct implementation.

$$\llbracket [\mathsf{v} \mathbin{/} \mathsf{x}] \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = \left[\llbracket \mathsf{v} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \Big/ \mathbf{x} \right]$$

Optimisation We could optimise Rule ($\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Deref) as follows:

- rename the current expressions except dereferencing to b;
- reform expressions both in $\mathsf{L^\tau}$ and $\mathbf{L^P}$ as $e ::= b \mid let\ x = b\ in\ e \mid !b$. In the case of $\mathbf{L^P}$ it would be $\cdots \mid \mathbf{!b\ with\ b}$.

  This allows expressions to compute e.g., pairs and projections.
- rewrite the Rule ($\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$-Deref) case for compiling $\mathsf{!b}$ into:

  $\mathbf{let\ x} = \llbracket \mathsf{b} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{in\ let\ x1 = x.1\ in\ let\ x2 = x.2\ in\ !x1\ with\ x2}$.
- as expressions execute atomically, this would also scale to the compiler for concurrent languages defined in later sections.

We do not use this approach to avoid nonstandard constructs.

4.1 Properties of the $\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$ Compiler

Theorem 2 (Compiler $\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$ is CC)**.** $\vdash \llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} : CC$

Theorem 3 (Compiler $\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$ is RSC)**.** $\vdash \llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} : RSC$

4.2 Back-translation from $\mathbf{L^P}$ to $\mathsf{L^U}$

4.2.1 Values Backtranslation

Here is how values are back translated.

$$\boxed{\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \mathbf{v} \to \mathsf{v}}$$

$$\langle\!\langle \mathbf{0} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \mathsf{true}$$

$$
\begin{aligned}
\langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \mathsf{false} && \text{if } \mathbf{n} \neq \mathbf{0}\\
\langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \mathsf{n} && \text{where } \mathsf{n} \approx_\beta \mathbf{n}\\
\langle\!\langle \mathbf{k} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \mathsf{0} &&\\
\langle\!\langle \langle \mathbf{v}, \mathbf{v}' \rangle \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \left\langle \langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}, \langle\!\langle \mathbf{v}' \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \right\rangle &&\\
\langle\!\langle \langle \mathbf{n}, \mathbf{v} \rangle \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \ell && \text{where } \ell \approx_\beta \langle \mathbf{n}, \mathbf{v} \rangle
\end{aligned}
$$

The backtranslation is nondeterministic, as $\approx_\beta$ is not injective. In this case we cannot make it injective (in the next compiler we can index it by types and make it so but here we do not have them). This is the reason why the backtranslation algorithm returns a set of contexts, as backtranslating an action that performs $\mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?$ could result in either $\mathtt{call}\ \mathsf{f}\ \mathsf{true}\ \mathsf{H}?$ or $\mathtt{call}\ \mathsf{f}\ \mathsf{0}\ \mathsf{H}?$. Now depending on f's body, which is the component to be compiled, supplying true or $\mathsf{0}$ may have different outcomes. Let us assume that the compilation of f, when receiving $\mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?$ does not get stuck. If f contains $\mathsf{if}\ \mathsf{x}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s}'$, supplying $\mathsf{0}$ will make it stuck. However, because we generate all possible contexts, we know that we generate also the context that will not cause f to be stuck. This is captured in Lemma 1 below.

Lemma 1 (Compiled code steps imply existence of source steps)**.**

$$
\begin{aligned}
&\forall\\
\text{if }\ &\Omega'' \approx_\beta \mathbf{\Omega''}\\
&\mathbf{\Omega''} \xRightarrow{\alpha?} \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}; \mathbf{s}'\rho\\
&\mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}; \mathbf{s}'\rho \xRightarrow{\alpha!} \mathbf{\Omega'}\\
&\{\alpha?\} = \{\alpha? \ \mid\ \alpha? \approx_\beta \alpha?\}\\
&\{\rho\} = \{\rho \ \mid\ \rho \approx_\beta \rho\}\\
\text{then }\ &\exists \alpha_{\mathsf{j}}? \in \{\alpha?\}, \rho_{\mathsf{y}} \in \{\rho\}, \mathsf{C_j}, \mathsf{H_j}, \mathsf{s_j}; \mathsf{s'_j}\rho'.\\
\text{if }\ &\Omega'' \xRightarrow{\alpha_{\mathsf{j}}?} \mathsf{C_j}, \mathsf{H_j} \triangleright \mathsf{s_j}; \mathsf{s'_j}\rho'\\
\text{then }\ &\mathsf{C_j}, \mathsf{H_j} \triangleright \mathsf{s_j}; \mathsf{s'_j}\rho_{\mathsf{y}} \approx_\beta \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}; \mathbf{s}'\rho\\
&\mathsf{C_j}, \mathsf{H_j} \triangleright \mathsf{s_j}; \mathsf{s'_j}\rho_{\mathsf{y}} \xRightarrow{\alpha!} \Omega'\\
&\alpha! \approx_\beta \alpha!\\
&\Omega' \approx_\beta \mathbf{\Omega'}
\end{aligned}
$$

4.2.2 Skeleton

$$\boxed{\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \mathbf{I} \to \mathsf{F}}$$

$$\langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \mathsf{f(x)} \mapsto \mathsf{incrementCounter(); return;} \qquad (\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-fun})$$

Functions call $\mathsf{incrementCounter()}$ before returning to ensure that when a returnback is modelled, the counter is incremented right before returning and not beforehand, as doing so would cause the possible execution of other bac-translated code blocks. Its implementation is described below.

$$\boxed{\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \overline{\mathbf{I}} \to \mathsf{A}}$$

$$\begin{aligned}
\langle\!\langle \overline{\mathbf{I}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = {} & \ell_i \mapsto 1; && (\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-skel})\\
& \ell_{glob} \mapsto 0 \\
& \mathsf{main(x)} \mapsto \mathsf{incrementCounter(); return;} \\
& \mathsf{incrementCounter()} \mapsto \text{see below} \\
& \mathsf{register(x)} \mapsto \text{see below} \\
& \mathsf{update(x)} \mapsto \text{see below} \\
& \langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} && \forall \mathbf{f} \in \overline{\mathbf{I}}
\end{aligned}$$

We assume compiled code does not implement functions $\mathsf{incrementCounter}$, $\mathsf{register}$ and update, they could be renamed to not generate conflicts if they were.

The skeleton sets up the infrastructure. It allocates global locations ℓ_{i}, which is used as a counter to count steps in actions, and ℓ_{glob}, which is used to keep track of attacker knowledge, as described below. Then it creates a dummy for all functions expected in the interfaces $\overline{\mathbf{I}}$ as well as a dummy for the main. Dummy functions return their parameter variable and they increment the global counter before that for reasons explained later.

4.2.3 Single Action Translation

We use the shortcut $\mathbf{ak}$ to indicate a list of pairs of locations and tag to access them $\overline{\langle \mathbf{n}, \eta \rangle}$ that is what the context has access to. We use functions `.loc` to access obtain all locations of such a list and `.cap` to obtain all the capabilities (or $\mathbf{0}$ when $\eta = \bot$) of the list.

We use function $\mathsf{incrementCounter}$ to increment the contents of ℓ_{i} by one.

$$\begin{aligned}
&\mathsf{incrementCounter(\ )} \mapsto \\
&\ \mathsf{let\ c} = !\ell_{\mathsf{i}}\ \mathsf{in\ let\ l} = \ell_i\ \mathsf{in\ l} := \mathsf{c} + 1
\end{aligned}$$

Starting from location ℓ_{g} we keep a list whose elements are pairs locations-numbers, we indicate this list as $\mathsf{L_{glob}}$.

We use function $\mathsf{register(\langle \ell, n \rangle)}$ which adds the pair $\langle \ell, \mathsf{n} \rangle$ to the list $\mathsf{L_{glob}}$. Any time we use this we are sure we are adding a pair for which no other existing pair in $\mathsf{L_{glob}}$ has a second projection equal to n. This function can be defined as follows:

$$
\begin{array}{l}
\mathsf{register}(\mathsf{x}) \mapsto \\
\ \mathsf{let}\ \mathsf{xl} = x.1 \\
\ \ \mathsf{in}\ \mathsf{let}\ \mathsf{xn} = x.2 \\
\ \ \ \mathsf{in}\ \mathsf{L_{glob}} :: \langle \mathsf{xl}, \mathsf{xn} \rangle
\end{array}
$$

$\mathsf{L_{glob}}$ is a list of pair elements, so it is implemented as a pair whose first projection is an element (a pair) and its second projection is another list; the empty list being $\mathsf{0}$. Where $::$ is a recursive function that starts from ℓ_{glob} and looks for its last element (i.e., it performs second projections until it hits a $\mathsf{0}$), then replaces that second projection with $\langle \langle \mathsf{xl}, \mathsf{xn} \rangle, \mathsf{0} \rangle$

Lemma 2 ($\mathsf{register}(\ell, \mathsf{n})$ does not add duplicates for n)**.** For n supplied as parameter by $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$, $\mathsf{C}; \mathsf{H} \triangleright \mathsf{register}(\ell, \mathsf{n}) \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H}' \triangleright \mathsf{skip}$ and $\langle \_, \mathsf{n} \rangle \notin \mathsf{L_{glob}}$

Proof. Simple analysis of Rules ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-call) to ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-ret-loc). □

We use function $\mathsf{update}(\mathsf{n}, \mathsf{v})$ which accesses the elements in the $\mathsf{L_{glob}}$ list, then takes the second projection of the element: if it is n it updates the first projection to v, otherwise it continues its recursive call. If it does not find an element for n, it gets stuck

Lemma 3 ($\mathsf{update}(\mathsf{n}, \mathsf{v})$ never gets stuck)**.** $\mathsf{C}; \mathsf{H} \triangleright \mathsf{update}(\mathsf{n}, \mathsf{v}) \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H}' \triangleright \mathsf{skip}$ for n and v supplied as parameters by $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ and $\mathsf{H}'{=}\mathsf{H}[\ell \mapsto \mathsf{v} \ / \ \ell \mapsto \_]$ for $\ell \approx_\beta \langle \mathbf{n}, \_ \rangle$.

Proof. Simple analysis of Rules ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-call) and ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-retback). □

We use the meta-level function $\mathtt{reachable}(\mathbf{H}, \mathbf{v}, \mathbf{ak})$ that returns a set of pairs $\langle \mathbf{n} \mapsto \mathbf{v} : \eta, \mathsf{e} \rangle$ such that all locations in are reachable from $\mathbf{H}$ starting from any location in $\mathbf{ak} \cup \mathbf{v}$ and that are not already in $\mathbf{ak}$ and such that e is a sequence of source-level instructions that evaluate to ℓ such that $\ell \approx_\beta \langle \mathbf{n}, \_ \rangle$.

Definition 15 (Reachable)**.**

$$
\mathtt{reachable}(\mathbf{H}, \mathbf{v}, \mathbf{ak}) = \left\{ \langle \mathbf{n} \mapsto \mathbf{v} : \mathbf{k}, \mathsf{e} \rangle \;\middle|\; \begin{array}{l}
\mathbf{n} \in \mathtt{reach}(\mathbf{n_{st}}, \mathbf{k_{st}}, \mathbf{H}) \\
\text{where } \mathbf{n_{st}} \in \mathbf{v} \cup \mathbf{ak}.\mathtt{loc} \\
\text{and } \mathbf{k_{st}} \in \mathbf{k_{root}} \cup \mathbf{ak}.\mathtt{cap} \\
\text{and } \mathbf{n} \mapsto \mathbf{v} : \mathbf{k} \in \mathbf{H} \\
\text{and } \mathbf{H} \triangleright !\mathbf{e} \hookrightarrow\!\!\!\rightarrow !\mathbf{n}\ \mathbf{with}\ \mathbf{k} \\
\text{and } \forall \mathsf{H}.\, \mathsf{H} \approx_\beta \mathbf{H} \\
\mathsf{H} \triangleright \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \hookrightarrow\!\!\!\rightarrow \ell \\
\text{and } (\ell, \mathbf{n}, \mathbf{k}) \in \beta
\end{array} \right\}
$$

Intuitively, `reachable`$(\cdot)$ finds out which new locations have been allocated by the compiled component and that are now reachable by the attacker (the first projection of the pair, $\mathbf{n} \mapsto \mathbf{v} : \eta$). Additionally, it tells how to reach those locations in the source so that we can $\mathsf{register}(\cdot)$ them for the source attacker (the backtranslated context) to access.

In this case we know by definition that $\mathbf{e}$ can only contain one $!$ and several $\cdot\mathbf{.1}$ or $\cdot\mathbf{.2}$. The base case for values is as before.

$$\boxed{\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \mathbf{e} \to \mathsf{e}}$$

$$\begin{aligned}
\langle\!\langle \mathbf{!e} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= !\langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\
\langle\!\langle \mathbf{e.1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\mathsf{.1} \\
\langle\!\langle \mathbf{e.2} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} &= \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\mathsf{.2}
\end{aligned}$$

The next function takes the following inputs: an action, its index, the previous function's heap, the previous attacker knowledge and the stack of functions called so far. It returns a set of: code, the new attacker knowledge, its heap, the stack of functions called and the function where the code must be put. In the returned parameters, the attacker knowledge, the heap and the stack of called functions serve as input to the next call.

$$\boxed{\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \alpha \times n \in \mathbb{N} \times \mathbf{H} \times \overline{\mathbf{n} \times \eta} \times \bar{\mathsf{f}} \to \left\{\mathsf{s} \times \overline{\mathbf{n} \times \eta} \times \mathbf{H} \times \bar{\mathsf{f}} \times \mathsf{f}\right\}}$$

$$\left\langle\!\!\left\langle \begin{matrix} \mathtt{call\ f\ v\ H?}, \\ n, \mathbf{H_{pre}}, \mathbf{ak}, \bar{\mathsf{f}} \end{matrix} \right\rangle\!\!\right\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \left\{ \begin{array}{l|l}
\left(\begin{array}{l}
\mathsf{if}\ !\ell_i == n\ \mathsf{then} \\
\quad \mathsf{incrementCounter}() \\
\quad \mathsf{let\ x1\ = new\ \ v_1\ in\ register}(\langle \mathsf{x1}, \mathsf{n_1} \rangle) \\
\quad \cdots \\
\quad \mathsf{let\ xj\ = new\ \ v_j\ in\ register}(\langle \mathsf{xj}, \mathsf{n_j} \rangle) \\
\quad \mathsf{update}(\mathsf{m_1}, \mathsf{u_1}) \\
\quad \cdots \\
\quad \mathsf{update}(\mathsf{m_l}, \mathsf{u_l}) \\
\quad \mathsf{call\ f\ v} \\
\mathsf{else\ skip}
\end{array} \right) &
\begin{array}{l}
\forall \\
\mathsf{v_1} = \langle\!\langle \mathbf{v_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\
\cdots \\
\mathsf{v_j} = \langle\!\langle \mathbf{v_j} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\
\mathsf{u_1} = \langle\!\langle \mathbf{u_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\
\cdots \\
\mathsf{u_l} = \langle\!\langle \mathbf{u_l} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\
\mathsf{v} = \langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}
\end{array} \\
, \mathbf{ak'}, \mathbf{H}, \mathsf{f}; \bar{\mathsf{f}}, \mathsf{f'} &
\end{array} \right\}$$

$$(\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-call})$$

$$\begin{aligned}
\text{where } \mathbf{H} \setminus \mathbf{H_{pre}} &= \mathbf{H_n} \\
\mathbf{H_n} &= \mathbf{n_1} \mapsto \mathbf{v_1} : \eta_1, \cdots, \mathbf{n_j} \mapsto \mathbf{v_j} : \eta_\mathbf{j} \\
\text{and } \mathbf{H} \cap \mathbf{H_{pre}} &= \mathbf{H_c} \\
\mathbf{H_c} &= \mathbf{m_1} \mapsto \mathbf{u_1} : \eta'_1, \cdots, \mathbf{m_l} \mapsto \mathbf{u_l} : \eta'_\mathbf{l} \\
\text{and } \mathbf{ak'} &= \mathbf{ak}, \langle \mathbf{n_1}, \eta_1 \rangle, \cdots, \langle \mathbf{n_j}, \eta_\mathbf{j} \rangle
\end{aligned}$$

and $\bar{\mathsf{f}} = \mathsf{f}'\overline{\mathsf{f}'}$

$$\left\langle\!\!\left\langle \begin{matrix} \mathtt{call\ f\ v\ \mathbf{H}!}, \\ n, \mathbf{H_{pre}}, \mathbf{ak}, \bar{\mathsf{f}} \end{matrix} \right\rangle\!\!\right\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \left\{ \begin{pmatrix} \mathsf{if}\ !\ell_i == n\ \mathsf{then} \\ \quad \mathsf{incrementCounter}() \\ \quad \mathsf{let\ l1} = \mathsf{e_1}\ \mathsf{in\ register}(\langle \mathsf{l1}, \mathsf{n_1} \rangle) \\ \quad \cdots \\ \quad \mathsf{let\ lj} = \mathsf{e_j}\ \mathsf{in\ register}(\langle \mathsf{lj}, \mathsf{n_j} \rangle) \\ \mathsf{else\ skip} \end{pmatrix}, \mathbf{ak}', \mathbf{H}, \mathsf{f}; \bar{\mathsf{f}}, \mathsf{f} \right\}$$

$$(\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-callback-loc})$$

if $\mathtt{reachable}(\mathbf{H}, \mathbf{v}, \mathbf{ak}) = \langle \mathbf{n_1} \mapsto \mathbf{v_1} : \eta_1, \mathsf{e_1} \rangle, \cdots, \langle \mathbf{n_j} \mapsto \mathbf{v_j} : \eta_j, \mathsf{e_j} \rangle$

and $\mathbf{ak}' = \mathbf{ak}, \langle \mathbf{n_1}, \eta_1 \rangle, \cdots, \langle \mathbf{n_j}, \eta_j \rangle$

$$\left\langle\!\!\left\langle \begin{matrix} \mathtt{ret\ \mathbf{H}?}, \\ n, \mathbf{H_{pre}}, \mathbf{ak}, \mathsf{f}; \bar{\mathsf{f}} \end{matrix} \right\rangle\!\!\right\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \left\{ \begin{matrix} \begin{pmatrix} \mathsf{if}\ !\ell_i == n\ \mathsf{then} \\ \quad // \textit{no}\ \mathsf{incrementCounter}()\ \textit{as explained} \\ \quad \mathsf{let\ x1} = \mathsf{new}\ \ \mathsf{v_1}\ \mathsf{in\ register}(\langle \mathsf{x1}, \mathsf{n_1} \rangle) \\ \quad \cdots \\ \quad \mathsf{let\ xj} = \mathsf{new}\ \ \mathsf{v_j}\ \mathsf{in\ register}(\langle \mathsf{xj}, \mathsf{n_j} \rangle) \\ \quad \mathsf{update}(\mathsf{m_1}, \mathsf{u_1}) \\ \quad \cdots \\ \quad \mathsf{update}(\mathsf{m_l}, \mathsf{u_l}) \\ \mathsf{else\ skip} \end{pmatrix} \\ , \mathbf{ak}', \mathbf{H}, \bar{\mathsf{f}}, \mathsf{f} \end{matrix} \middle| \begin{matrix} \forall \\ \mathsf{v_1} = \langle\!\langle \mathbf{v_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\ \cdots \\ \mathsf{v_j} = \langle\!\langle \mathbf{v_j} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\ \mathsf{u_1} = \langle\!\langle \mathbf{u_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\ \cdots \\ \mathsf{u_l} = \langle\!\langle \mathbf{u_l} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \end{matrix} \right\}$$

$$(\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-retback})$$

where $\mathbf{H} \setminus \mathbf{H_{pre}} = \mathbf{H_n}$

$\mathbf{H_n} = \mathbf{n_1} \mapsto \mathbf{v_1} : \eta_1, \cdots, \mathbf{n_j} \mapsto \mathbf{v_j} : \eta_j$

and $\mathbf{H} \cap \mathbf{H_{pre}} = \mathbf{H_c}$

$\mathbf{H_c} = \mathbf{m_1} \mapsto \mathbf{u_1} : \eta'_1, \cdots, \mathbf{m_l} \mapsto \mathbf{u_l} : \eta'_l$

and $\mathbf{ak}' = \mathbf{ak}, \langle \mathbf{n_1}, \eta_1 \rangle, \cdots, \langle \mathbf{n_j}, \eta_j \rangle$

$$\left\langle\!\!\left\langle \begin{matrix} \mathtt{ret\ \mathbf{H}!}, \\ n, \mathbf{H_{pre}}, \mathbf{ak}, \mathsf{f}; \bar{\mathsf{f}} \end{matrix} \right\rangle\!\!\right\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \left\{ \begin{pmatrix} \mathsf{if}\ !\ell_i == n\ \mathsf{then} \\ \quad \mathsf{incrementCounter}() \\ \quad \mathsf{let\ l1} = \mathsf{e_1}\ \mathsf{in\ register}(\langle \mathsf{l1}, \mathsf{n_1} \rangle) \\ \quad \cdots \\ \quad \mathsf{let\ lj} = \mathsf{e_j}\ \mathsf{in\ register}(\langle \mathsf{lj}, \mathsf{n_j} \rangle) \\ \mathsf{else\ skip} \end{pmatrix}, \mathbf{ak}', \mathbf{H}, \bar{\mathsf{f}}, \mathsf{f}' \right\}$$

$$(\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-ret-loc})$$

$$\begin{aligned}
&\text{if } \texttt{reachable}(\mathbf{H}, \mathbf{0}, \mathbf{ak}) = \langle \mathbf{n_1} \mapsto \mathbf{v_1} : \eta_1, \mathsf{e_1}\rangle, \cdots, \langle \mathbf{n_j} \mapsto \mathbf{v_j} : \eta_j, \mathsf{e_j}\rangle\\
&\text{and } \mathbf{ak}' = \mathbf{ak}, \langle \mathbf{n_1}, \eta_1\rangle, \cdots, \langle \mathbf{n_j}, \eta_j\rangle\\
&\text{and } \overline{\mathsf{f}} = \mathsf{f}'\overline{\mathsf{f}'}
\end{aligned}$$

This is the back-translation of functions. Each action is wrapped in an if statement checking that the action to be mimicked is that one (the same function may behave differently if called twice and we need to ensure this). After the if, the counter checking for the action index ℓ_{i} is incremented. This is not done in case of a return immediately, but only just before the return itself, so the increment is added in the skeleton already. (there could be a callback to the same function after the return and then we wouldn't return but execute the callback code instead)

When back-translating a ?-decorated, we need to set up the heap correctly before the call itself. That means calculating the new locations that this action allocated ($\mathbf{H_n}$), allocating them and registering them in the $\mathsf{L_{glob}}$ list via the $\mathsf{register}(\cdot)$ function. These locations are also added to the attacker knowledge $\mathbf{ak}'$. Then we need to update the heap locations we already know of. These locations are $\mathbf{H_c}$ and as we know them already, we use the $\mathsf{update}(\cdot)$ function.

When back-translating a !-decorated action we need to calculate what part of the heap we can reach from there, and so we rely on the $\texttt{reachable}(\cdot)$ function to return a list of pairs of locations $\mathbf{n}$ and expressions e. We use $\mathbf{n}$ to expand the attacker knowledge $\mathbf{ak}'$ as these locations are now reachable. We use $\mathbf{e}$ to reach these locations in the source heap so that we can $\mathsf{register}$ them and ensure they are accessible through $\mathsf{L_{glob}}$.

Finally, we use parameter $\overline{\mathsf{f}}$ to keep track of the call stack, so making a call to f pushes f on the stack ($\mathsf{f}; \overline{\mathsf{f}}$) and making a return pops a stack $\mathsf{f}; \overline{\mathsf{f}}$ to $\overline{\mathsf{f}}$. That stack carries the information to instantiate the f in the return parameters, which is the location where the code needs to be allocated.

$$\boxed{\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \overline{\alpha} \times n \in \mathbb{N} \times \mathbf{H} \times \overline{\mathbf{n} \times \eta} \times \overline{\mathsf{f}} \to \{\overline{\mathsf{s}, \mathsf{f}}\}}$$

$$\langle\!\langle \varnothing \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \varnothing \qquad (\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-listact-b})$$

$$\langle\!\langle \alpha\overline{\alpha}, n, \mathbf{H_{pre}}, \mathbf{ak}, \overline{\mathsf{f}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \left\{ \mathsf{s}, \mathsf{f}; \overline{\mathsf{s}, \mathsf{f}} \;\middle|\; \begin{array}{l} \mathsf{s}, \mathbf{ak}', \mathbf{H}', \overline{\mathsf{f}'}, \mathsf{f} = \langle\!\langle \alpha, n, \mathbf{H_{pre}}, \mathbf{ak}, \overline{\mathsf{f}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\\ \overline{\mathsf{s}, \mathsf{f}} \in \langle\!\langle \overline{\alpha}, n+1, \mathbf{H}', \mathbf{ak}', \overline{\mathsf{f}'} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \end{array} \right\}$$

$$(\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-listact-i})$$

This recursive call ensures the parameters are passed around correctly. Note that each element in a set returned by the single-action back-translation has the same $\mathbf{ak}'$, $\mathbf{H}$ and f', the only elements that change are in the code s due to the backtranslation of values. Thus the recursive call can pass those parameters taken from any element of the set.

4.2.4 The Back-translation Algorithm $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$

$$\boxed{\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} : \overline{\mathbf{I}} \times \overline{\alpha} \to \{\mathsf{A}\}}$$

$$\langle\!\langle \overline{\mathbf{I}}, \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \left\{ \mathsf{A} \;\middle|\; \begin{array}{l} \mathsf{A} = \mathsf{A}_{\mathsf{skel}} \bowtie \overline{\mathsf{s},\mathsf{f}} \\ \text{for all } \overline{\mathsf{s},\mathsf{f}} \in \left\{\overline{\mathsf{s},\mathsf{f}}\right\} \\ \text{where } \left\{\overline{\mathsf{s},\mathsf{f}}\right\} = \langle\!\langle \overline{\alpha}, 1, \mathbf{H_0}, \varnothing, \mathsf{main} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\ \mathbf{H_0} = \mathbf{0} \mapsto \mathbf{0} : \mathbf{k_{root}} \\ \mathsf{A}_{\mathsf{skel}} = \langle\!\langle \overline{\mathbf{I}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \end{array} \right\} \qquad (\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-main})$$

This is the real back-translation algorithm: it calls the skeleton and joins it with each element of the set returned by the trace back-translation.

$$\boxed{\bowtie : \mathsf{A} \times \overline{\mathsf{s},\mathsf{f}} \to \mathsf{A}}$$

$$\mathsf{A} \bowtie \varnothing = \mathsf{A} \qquad (\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\text{-join})$$

$$\begin{aligned} \mathsf{H}; \mathsf{F}_1; \cdots; \mathsf{F}; \cdots; \mathsf{F}_\mathsf{n} \bowtie \overline{\mathsf{s},\mathsf{f}}; \mathsf{s}, \mathsf{f} &= \mathsf{H}; \mathsf{F}_1; \cdots; \mathsf{F}'; \cdots; \mathsf{F}_\mathsf{n} \bowtie \overline{\mathsf{s},\mathsf{f}} \\ \text{where } \mathsf{F} &= \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}'; \mathsf{return}; \\ \mathsf{F}' &= \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{s}'; \mathsf{return}; \end{aligned}$$

When joining we add from the last element of the list so that the functions we create have the concatenation of if statements (those guarded by the counter on ℓ_i) that are sorted (guards with a test for $\ell_\mathsf{i} = 4$ are before those with a test $\ell_\mathsf{i} = 5$).

4.2.5 Correctness of the Back-translation

Theorem 4 ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ is correct)**.**

$$\begin{array}{c} \forall \\ \text{if } \mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right]\right) \xRightarrow{\overline{\alpha}} \mathbf{\Omega} \\ \mathbf{\Omega} \xRightarrow{\epsilon} \mathbf{\Omega}' \\ \overline{\mathbf{I}} = \mathtt{names}(\mathbf{A}) \\ \overline{\alpha} \equiv \overline{\alpha'} \cdot \alpha? \\ \ell_\mathsf{i}; \ell_{\mathsf{glob}} \notin \beta \\ \text{then } \exists \mathsf{A} \in \langle\!\langle \mathbf{I}, \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \\ \text{such that } \ \Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \Omega \\ \text{and } \overline{\alpha} \approx_\beta \overline{\alpha} \end{array}$$

$$\Omega \approx_{\beta} \mathbf{\Omega}$$
$$\Omega.\mathsf{H}.\ell_{\mathsf{i}} = \|\overline{\alpha}\| + 1$$

The back-translation is correct if it takes a target attacker that will reduce to a state together with a compiled component and it produces a set of source attackers such that one of them, that together with the source component will reduce to a related state performing related actions. Also it needs to ensure the step is incremented correctly.

4.2.6 Remark on the Backtranslation

Some readers may wonder whether the hassle of setting up a source-level representation of the whole target heap is necessary. Indeed for those locations that are allocated by the context, this is not. If we changed the source semantics to have an oracle that predicts what a $\mathsf{let\ x = new\ e\ in\ s}$ statement will return as the new location, we could simplify this. In fact, currently the backtranslation stores target locations in the list $\mathsf{L_{glob}}$ and looks them up based on their target name, as it does not know what source name will be given to them. The oracle would obviate this problem, so we could hard code the name of these locations, knowing exactly the identifier that will be returned by the allocator. For the functions to be correct in terms of syntax, we would need to pre-emptively allocate all the locations with that identifier so that their names are in scope and they can be referred to.

However, the problem still persists for locations created by the component, as their names cannot be hard coded, as they are not in scope. Thus we would still require reach to reach these locations, register to add them to the list and update to update their values in case the attacker does so.

Thus we simplify the scenario and stick to a more standard, oracle-less semantics and to a generalised approach to location management in the backtranslation.

5 The Source Language: L^{τ}

This is an imperative, concurrent while language with monitors.

$$
\begin{aligned}
\textit{Whole Programs } \mathsf{P} &::= \Delta; \mathsf{H}; \overline{\mathsf{F}}; \overline{\mathsf{I}} \\
\textit{Components } \mathsf{C} &::= \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}} \\
\textit{Contexts } \mathsf{A} &::= \mathsf{H}; \overline{\mathsf{F}}\,[\cdot] \\
\textit{Interfaces } \mathsf{I} &::= \mathsf{f} \\
\textit{Functions } \mathsf{F} &::= \mathsf{f}(\mathsf{x} : \tau) \mapsto \mathsf{s}; \mathsf{return}; \\
\textit{Operations } \oplus &::= + \mid - \\
\textit{Comparison } \otimes &::= == \mid < \mid > \\
\textit{Values } \mathsf{v} &::= \mathsf{b} \in \{\mathsf{true}, \mathsf{false}\} \mid \mathsf{n} \in \mathbb{N} \mid \langle \mathsf{v}, \mathsf{v} \rangle \mid \ell \\
\textit{Expressions } \mathsf{e} &::= \mathsf{x} \mid \mathsf{v} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e} \otimes \mathsf{e} \mid !\mathsf{e} \mid \langle \mathsf{e}, \mathsf{e} \rangle \mid \mathsf{e.1} \mid \mathsf{e.2} \\
\textit{Statements } \mathsf{s} &::= \mathsf{skip} \mid \mathsf{s}; \mathsf{s} \mid \mathsf{let}\ \mathsf{x} : \tau = \mathsf{e}\ \mathsf{in}\ \mathsf{s} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s} \\
&\quad \mid \mathsf{x} := \mathsf{e} \mid \mathsf{let}\ \mathsf{x} = \mathsf{new}_{\tau}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \mid \mathsf{call}\ \mathsf{f}\ \mathsf{e} \\
&\quad \mid (\Vert\ \mathsf{s}) \mid \mathsf{endorse}\ \mathsf{x} = \mathsf{e}\ \mathsf{as}\ \varphi\ \mathsf{in}\ \mathsf{s} \\
\textit{Types } \tau &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \tau \times \tau \mid \mathsf{Ref}\ \tau \mid \mathsf{UN} \\
\textit{Superficial Types } \varphi &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{UN} \times \mathsf{UN} \mid \mathsf{Ref}\ \mathsf{UN} \\
\textit{Eval. Ctxs. } \mathsf{E} &::= [\cdot] \mid \mathsf{e} \oplus \mathsf{E} \mid \mathsf{E} \oplus \mathsf{n} \mid \mathsf{e} \otimes \mathsf{E} \mid \mathsf{E} \otimes \mathsf{n} \\
&\quad \mid !\mathsf{E} \mid \langle \mathsf{e}, \mathsf{E} \rangle \mid \langle \mathsf{E}, \mathsf{v} \rangle \mid \mathsf{E.1} \mid \mathsf{E.2} \\
\textit{Heaps } \mathsf{H} &::= \varnothing \mid \mathsf{H}; \ell \mapsto \mathsf{v} : \tau \\
\textit{Monitors } \mathsf{M} &::= (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c) \\
\textit{Mon. States } \sigma &\in \mathcal{S} \\
\textit{Mon. Reds. } \rightsquigarrow &::= \varnothing \mid \rightsquigarrow; (\mathsf{s}, \mathsf{s}) \\
\textit{Environments } \Gamma, \Delta &::= \varnothing \mid \Gamma; (\mathsf{x} : \tau) \\
\textit{Store Env. } \Delta &::= \varnothing \mid \Delta; (\ell : \tau) \\
\textit{Substitutions } \rho &::= \varnothing \mid \rho[\mathsf{v} \,/\, \mathsf{x}] \\
\textit{Processes } \pi &::= (\mathsf{s})_{\overline{\mathsf{f}}} \\
\textit{Soups } \Pi &::= \varnothing \mid \Pi \Vert \pi \\
\textit{Prog. States } \Omega &::= \mathsf{C}, \mathsf{H} \triangleright \Pi \\
\textit{Labels } \lambda &::= \epsilon \mid \alpha \\
\textit{Actions } \alpha &::= \texttt{call f v?} \mid \texttt{call f v!} \mid \texttt{ret !} \mid \texttt{ret ?} \\
\textit{Traces } \overline{\alpha} &::= \varnothing \mid \overline{\alpha} \cdot \alpha
\end{aligned}
$$

We highlight elements that have changed from L^{U}.

5.1 Static Semantics of L^{τ}

The static semantics follows these typing judgements.

$\vdash \mathsf{C} : \mathsf{UN}$	Component C is well-typed.
$\mathsf{C} \vdash \mathsf{F} : \tau$	Function F takes arguments of type τ under component C.
$\Delta, \Gamma \vdash \diamond$	Environments Γ and Δ are well-formed.
$\Delta \vdash \mathsf{ok}$	Environment Δ is safe.
$\tau \vdash \circ$	Type τ is insecure.
$\Delta, \Gamma \vdash \mathsf{e} : \tau$	Expression e has type τ inΓ.
$\mathsf{C}, \Delta, \Gamma \vdash \mathsf{s}$	Statement s is well-typed in C and Γ.
$\mathsf{C}, \Delta, \Gamma \vdash \pi$	Single process π is well-typed in C and Γ.
$\mathsf{C}, \Delta, \Gamma \vdash \Pi$	Soup Π is well-typed in C and Γ.
$\vdash \mathsf{H} : \Delta$	Heap H respects the typing of Δ.
$\vdash \mathsf{M}$	Monitor M is valid.

5.1.1 Auxiliary Functions

We rely on these standard auxiliary functions: $\texttt{names}(\cdot)$ extracts the defined names (e.g., function and interface names). $\texttt{fv}(\cdot)$ returns free variables while $\texttt{fn}(\cdot)$ returns free names (i.e., a call to a defined function). $\texttt{dom}(\cdot)$ returns the domain of a particular element (e.g., all the allocated locations in a heap). We denote access to the parts of C and P via functions $.\texttt{funs}$, $.\texttt{intfs}$ and $.\texttt{mon}$. We denote access to parts of M with a dot notation, so $\mathsf{M}.\Delta$ means Δ where $\mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c)$.

5.1.2 Typing Rules

$$\boxed{\vdash \mathsf{C}}$$

$$\frac{\mathsf{C} \equiv \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}} \qquad \mathsf{C} \vdash \overline{\mathsf{F}} : \mathsf{UN} \qquad \texttt{names}(\overline{\mathsf{F}}) \cap \texttt{names}(\overline{\mathsf{I}}) = \varnothing \qquad \Delta \vdash \mathsf{ok}}{\vdash \mathsf{C} : \mathsf{UN}}\ (\mathsf{TL}^{\tau}\text{-component})$$

$$\boxed{\mathsf{C} \vdash \mathsf{F} : \mathsf{UN}}$$

$$\frac{\begin{array}{c}\mathsf{F} \equiv \mathsf{f}(\mathsf{x} : \mathsf{UN}) \mapsto \mathsf{s}; \mathsf{return}; \qquad \mathsf{C}, \Delta; \mathsf{x} : \mathsf{UN} \vdash \mathsf{s} \\ \mathsf{C} \equiv \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}} \qquad \forall \mathsf{f} \in \texttt{fn}(s), \mathsf{f} \in \texttt{dom}(\mathsf{C}.\texttt{funs}) \vee \mathsf{f} \in \texttt{dom}(\mathsf{C}.\texttt{intfs})\end{array}}{\mathsf{C} \vdash \mathsf{F} : \mathsf{UN}}\ (\mathsf{TL}^{\tau}\text{-function})$$

$$\boxed{\Delta, \Gamma \vdash \diamond}$$

(TL$^\tau$-env-e)
$$\frac{}{\varnothing;\varnothing \vdash \diamond}$$

(TL$^\tau$-env-var)
$$\frac{\Delta,\Gamma \vdash \diamond \qquad \mathsf{x} \notin \mathtt{dom}(\Gamma)}{\Delta,\Gamma;(\mathsf{x}:\tau) \vdash \diamond}$$

(TL$^\tau$-env-loc)
$$\frac{\Delta,\Gamma \vdash \diamond \qquad \mathsf{l} \notin \mathtt{dom}(\Delta)}{\Delta;(\mathsf{l}:\tau);\Gamma \vdash \diamond}$$

$$\boxed{\Delta,\Gamma \vdash \mathsf{ok}}$$

(TL$^\tau$-safe-e)
$$\frac{}{\varnothing \vdash \mathsf{ok}}$$

(TL$^\tau$-safe-loc)
$$\frac{\Delta \vdash \mathsf{ok} \qquad \mathsf{l} \notin \mathtt{dom}(\Gamma) \qquad \mathsf{UN} \notin \tau}{\Gamma;(\mathsf{l}:\tau) \vdash \mathsf{ok}}$$

$$\boxed{\Delta,\Gamma \vdash \mathsf{UN}}$$

(TL$^\tau$-env-e)
$$\frac{}{\varnothing \vdash \mathsf{UN}}$$

(TL$^\tau$-env-var)
$$\frac{\Delta,\Gamma \vdash \mathsf{UN} \qquad \mathsf{x} \notin \mathtt{dom}(\Gamma)}{\Gamma,(\mathsf{x}:\mathsf{UN}) \vdash \mathsf{UN}}$$

(TL$^\tau$-env-loc)
$$\frac{\Delta,\Gamma \vdash \mathsf{UN} \qquad \mathsf{l} \notin \mathtt{dom}(\Gamma)}{\Gamma,(\mathsf{l}:\mathsf{UN}) \vdash \mathsf{UN}}$$

$$\boxed{\tau \vdash \circ}$$

(TL$^\tau$-bool-pub)
$$\frac{}{\mathsf{Bool} \vdash \circ}$$

(TL$^\tau$-nat-pub)
$$\frac{}{\mathsf{Nat} \vdash \circ}$$

(TL$^\tau$-pair-pub)
$$\frac{\tau \vdash \circ \qquad \tau' \vdash \circ}{\tau \times \tau' \vdash \circ}$$

(TL$^\tau$-un-pub)
$$\frac{}{\mathsf{UN} \vdash \circ}$$

(TL$^\tau$-references-pub)
$$\frac{}{\mathsf{Ref}\ \mathsf{UN} \vdash \circ}$$

$$\boxed{\Delta,\Gamma \vdash \mathsf{e}:\tau}$$

(TL$^\tau$-true)
$$\frac{\Delta,\Gamma \vdash \diamond}{\Delta,\Gamma \vdash \mathsf{true}:\mathsf{Bool}}$$

(TL$^\tau$-false)
$$\frac{\Delta,\Gamma \vdash \diamond}{\Delta,\Gamma \vdash \mathsf{false}:\mathsf{Bool}}$$

(TL$^\tau$-nat)
$$\frac{\Delta,\Gamma \vdash \diamond}{\Delta,\Gamma \vdash \mathsf{n}:\mathsf{Nat}}$$

(TL$^\tau$-var)
$$\frac{\mathsf{x}:\tau \in \Gamma}{\Delta,\Gamma \vdash \mathsf{x}:\tau}$$

(TL$^\tau$-loc)
$$\frac{\mathsf{l}:\tau \in \Delta}{\Delta,\Gamma \vdash \mathsf{l}:\mathsf{Ref}\ \tau}$$

(TL$^\tau$-pair)
$$\frac{\begin{array}{c}\Delta,\Gamma \vdash \mathsf{e}_1:\tau\\ \Delta,\Gamma \vdash \mathsf{e}_2:\tau'\end{array}}{\Delta,\Gamma \vdash \langle \mathsf{e}_1,\mathsf{e}_2\rangle:\tau\times\tau'}$$

(TL$^\tau$-proj-1)
$$\frac{\Delta,\Gamma \vdash \mathsf{e}:\tau\times\tau'}{\Delta,\Gamma \vdash \mathsf{e.1}:\tau}$$

(TL$^\tau$-proj-2)
$$\frac{\Delta,\Gamma \vdash \mathsf{e}:\tau\times\tau'}{\Delta,\Gamma \vdash \mathsf{e.2}:\tau'}$$

(TL$^\tau$-dereference)
$$\frac{\Delta,\Gamma \vdash \mathsf{e}:\mathsf{Ref}\ \tau}{\Delta,\Gamma \vdash !\mathsf{e}:\tau}$$

(TL$^\tau$-op)
$$\frac{\Delta,\Gamma \vdash \mathsf{e}:\mathsf{Nat} \qquad \Delta,\Gamma \vdash \mathsf{e}':\mathsf{Nat}}{\Delta,\Gamma \vdash \mathsf{e} \oplus \mathsf{e}':\mathsf{Nat}}$$

(TL$^\tau$-cmp)
$$\frac{\Delta,\Gamma \vdash \mathsf{e}:\mathsf{Nat} \qquad \Delta,\Gamma \vdash \mathsf{e}':\mathsf{Nat}}{\Delta,\Gamma \vdash \mathsf{e} \otimes \mathsf{e}':\mathsf{Bool}}$$

(TL$^\tau$-coercion)
$$\frac{\mathsf{C},\Delta,\Gamma \vdash \mathsf{e}:\tau \qquad \tau \vdash \circ}{\mathsf{C},\Delta,\Gamma \vdash \mathsf{e}:\mathsf{UN}}$$

$$\boxed{\mathsf{C},\Delta,\Gamma \vdash \mathsf{s}}$$

$$\overset{(\mathsf{TL}^{\tau}\text{-skip})}{\dfrac{}{C, \Delta, \Gamma \vdash \mathsf{skip}}}
\qquad
\overset{(\mathsf{TL}^{\tau}\text{-function-call})}{\dfrac{\begin{array}{c}((f \in \mathtt{dom}(C.\mathtt{funs})) \vee (f \in \mathtt{dom}(C.\mathtt{intfs}))) \\ \Delta, \Gamma \vdash e : \mathsf{UN}\end{array}}{\Delta, \Gamma \vdash \mathsf{call}\ f\ e}}$$

$$\overset{(\mathsf{TL}^{\tau}\text{-sequence})}{\dfrac{\begin{array}{c}C, \Delta, \Gamma \vdash s_u \\ C, \Delta, \Gamma \vdash s\end{array}}{C, \Delta, \Gamma \vdash s_u; s}}
\qquad
\overset{(\mathsf{TL}^{\tau}\text{-letin})}{\dfrac{\begin{array}{c}\Delta, \Gamma \vdash e : \tau \\ C, \Gamma; x : \tau \vdash s\end{array}}{C, \Delta, \Gamma \vdash \mathsf{let}\ x : \tau = e\ \mathsf{in}\ s}}
\qquad
\overset{(\mathsf{TL}^{\tau}\text{-assign})}{\dfrac{\begin{array}{c}\Delta, \Gamma \vdash x : \mathsf{Ref}\ \tau \\ \Delta, \Gamma \vdash e' : \tau\end{array}}{C, \Delta, \Gamma \vdash x := e'}}$$

$$\overset{(\mathsf{TL}^{\tau}\text{-new})}{\dfrac{\begin{array}{c}\Delta, \Gamma \vdash e : \tau \\ C, \Gamma; x : \mathsf{Ref}\ \tau \vdash s\end{array}}{C, \Delta, \Gamma \vdash \mathsf{let}\ x = \mathsf{new}_{\tau}\ e\ \mathsf{in}\ s}}
\qquad
\overset{(\mathsf{TL}^{\tau}\text{-if})}{\dfrac{\begin{array}{c}\Delta, \Gamma \vdash e : \mathtt{Bool} \\ C, \Delta, \Gamma \vdash s_t \quad C, \Delta, \Gamma \vdash s_e\end{array}}{C, \Delta, \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ s_t\ \mathsf{else}\ s_e}}$$

$$\overset{(\mathsf{TL}^{\tau}\text{-fork})}{\dfrac{C, \Delta, \Gamma \vdash s}{C, \Delta, \Gamma \vdash (\parallel s)}}
\qquad
\overset{(\mathsf{TL}^{\tau}\text{-endorse})}{\dfrac{\Delta, \Gamma \vdash e : \mathsf{UN} \quad C, \Delta, \Gamma; (x : \varphi) \vdash s}{C, \Delta, \Gamma \vdash \mathsf{endorse}\ x = e\ \mathsf{as}\ \varphi\ \mathsf{in}\ s}}$$

$$\boxed{C, \Delta, \Gamma \vdash \pi}$$

$$\overset{(\mathsf{TL}^{\tau}\text{-process})}{\dfrac{C, \Delta, \Gamma \vdash s}{C, \Delta, \Gamma \vdash (s)_{\bar{f}}}}$$

$$\boxed{C, \Delta, \Gamma \vdash \Pi}$$

$$\overset{(\mathsf{TL}^{\tau}\text{-soup})}{\dfrac{C, \Delta, \Gamma \vdash \pi \quad C, \Delta, \Gamma \vdash \Pi}{C, \Delta, \Gamma \vdash \pi \parallel \Pi}}$$

$$\boxed{\vdash H : \Delta}$$

$$\overset{(\mathsf{L}^{\tau}\text{-Heap-ok-i})}{\dfrac{\begin{array}{cc}\ell :\mapsto v : \tau \in H & \ell : \tau \in \Delta \\ \vdash H : \Delta & \Delta\varnothing \vdash v : \tau\end{array}}{\vdash H : \Delta; \ell : \tau}}
\qquad
\overset{(\mathsf{L}^{\tau}\text{-Heap-ok-b})}{\dfrac{}{\vdash H : \varnothing}}$$

$$\boxed{\vdash M}$$

$$\overset{(\mathsf{L}^{\tau}\text{-Monitor})}{\dfrac{M \equiv (\{s\}, \rightsquigarrow, s_0, \Delta, s_c) \qquad \forall s \exists s'.(s, s') \in \rightsquigarrow}{\vdash M}}$$

Notes Monitor typing just ensures that the monitor is coherent and that it can't get stuck for no good reason.

5.1.3 UN Typing

Attackers cannot have $\mathsf{new}_\tau\ \mathsf{t}$ terms where τ is different from UN.

$$\boxed{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN}}$$

(TUL$^\tau$-base)

$$\frac{\begin{array}{cc} \mathsf{A} = \mathsf{H}; \overline{\mathsf{F}}\,[\cdot] & \Delta \vdash_{\mathsf{UN}} \overline{\mathsf{F}} \\ \mathtt{dom}(\mathsf{H}) \cap \mathtt{dom}(\Delta) = \varnothing & \mathtt{dom}(\Delta) \cap (\mathtt{fv}(\overline{\mathsf{F}}) \cup \mathtt{fv}(\mathsf{H})) = \varnothing \end{array}}{\Delta \vdash_{\mathsf{UN}} \mathsf{A}}$$

(TUL$^\tau$-true)

$$\frac{\Delta, \Gamma \vdash \diamond}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{true} : \mathsf{UN}}$$

(TUL$^\tau$-false)

$$\frac{\Delta, \Gamma \vdash \diamond}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{false} : \mathsf{UN}}$$

(TUL$^\tau$-nat)

$$\frac{\Delta, \Gamma \vdash \diamond}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{n} : \mathsf{UN}}$$

(TUL$^\tau$-var)

$$\frac{\mathsf{x} : \tau \in \Gamma}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{x} : \mathsf{UN}}$$

(TUL$^\tau$-loc)

$$\frac{\mathsf{l} : \mathsf{UN} \notin \Delta}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{l} : \mathsf{UN}}$$

(TUL$^\tau$-pair)

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e}_1 : \mathsf{UN} \\ \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e}_2 : \mathsf{UN} \end{array}}{\Delta, \Gamma \vdash_{\mathsf{UN}} \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \mathsf{UN}}$$

(TUL$^\tau$-proj-1)

$$\frac{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN}}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e.1} : \mathsf{UN}}$$

(TUL$^\tau$-proj-2)

$$\frac{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN}}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e.2} : \mathsf{UN}}$$

(TUL$^\tau$-dereference)

$$\frac{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN}}{\Delta, \Gamma \vdash_{\mathsf{UN}} !\mathsf{e} : \mathsf{UN}}$$

(TUL$^\tau$-op)

$$\frac{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN} \qquad \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e}' : \mathsf{UN}}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} \oplus \mathsf{e}' : \mathsf{UN}}$$

(TUL$^\tau$-cmp)

$$\frac{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN} \qquad \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e}' : \mathsf{UN}}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} \otimes \mathsf{e}' : \mathsf{UN}}$$

$$\boxed{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s}}$$

(TUL$^\tau$-skip)

$$\frac{}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{skip}}$$

(TUL$^\tau$-function-call)

$$\frac{\begin{array}{c} ((\mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{funs})) \vee (\mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{intfs}))) \\ \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN} \end{array}}{\Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{call}\ \mathsf{f}\ \mathsf{e}}$$

(TUL$^\tau$-sequence)

$$\frac{\begin{array}{c} \mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s}_\mathsf{u} \\ \mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s} \end{array}}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s}_\mathsf{u}; \mathsf{s}}$$

(TUL$^\tau$-letin)

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN} \\ \mathsf{C}, \Gamma; \mathsf{x} : \mathsf{UN} \vdash_{\mathsf{UN}} \mathsf{s} \end{array}}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{let}\ \mathsf{x} : \mathsf{UN} = \mathsf{e}\ \mathsf{in}\ \mathsf{s}}$$

(TUL$^\tau$-assign)

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{x} : \mathsf{UN} \\ \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e}' : \mathsf{UN} \end{array}}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{x} := \mathsf{e}'}$$

(TUL$^\tau$-new)

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathsf{UN} \\ \mathsf{C}, \Gamma; \mathsf{x} : \mathsf{UN} \vdash_{\mathsf{UN}} \mathsf{s} \end{array}}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{let}\ \mathsf{x} = \mathsf{new}_{\mathsf{UN}}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}}$$

(TUL$^\tau$-if)

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{e} : \mathtt{Bool} \\ \mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s}_\mathsf{t} \qquad \mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s}_\mathsf{e} \end{array}}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}_\mathsf{t}\ \mathsf{else}\ \mathsf{s}_\mathsf{e}}$$

(TUL$^\tau$-fork)

$$\frac{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} \mathsf{s}}{\mathsf{C}, \Delta, \Gamma \vdash_{\mathsf{UN}} (\parallel \mathsf{s})}$$

5.2 Dynamic Semantics of L^{τ}

Function `mon-care`$(\cdot)$ returns the part of a heap the monitor cares for (Rule L^{τ}-Monitor-related heap). Rules L^{τ}-Jump-Internal to L^{τ}-Jump-OUT dictate the kind of a jump between two functions: if internal to the component/attacker, in(from the attacker to the component) or out(from the component to the attacker). Rule L^{τ}-Plug tells how to obtain a whole program from a component and an attacker. Rule L^{τ}-Initial State tells the initial state of a whole program. Rule L^{τ}-Initial-heap produces a heap that satisfies a Δ, initialised with base values. Rule L^{τ}-Monitor Step tells when a monitor makes a single step given a heap.

$$\boxed{\texttt{mon-care}(\cdot)}$$

(L$^{\tau}$-Monitor-related heap)

$$\frac{\mathsf{H}' = \{\ell \mapsto \mathsf{v} : \tau \mid \ell \mapsto \mathsf{v} : \tau \in \mathsf{H}\}}{\vdash \texttt{mon-care}(\mathsf{H}, \Delta) = \mathsf{H}'}$$

$$\boxed{\text{Helpers}}$$

(L$^{\tau}$-Jump-Internal)

$$\frac{\begin{array}{c}((\mathsf{f}' \in \bar{\mathsf{I}} \land \mathsf{f} \in \bar{\mathsf{I}}) \lor \\ (\mathsf{f}' \notin \bar{\mathsf{I}} \land \mathsf{f} \notin \bar{\mathsf{I}}))\end{array}}{\bar{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal}}$$

(L$^{\tau}$-Jump-IN)

$$\frac{\mathsf{f} \in \bar{\mathsf{I}} \land \mathsf{f}' \notin \bar{\mathsf{I}}}{\bar{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{in}}$$

(L$^{\tau}$-Jump-OUT)

$$\frac{\mathsf{f} \notin \bar{\mathsf{I}} \land \mathsf{f}' \in \bar{\mathsf{I}}}{\bar{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{out}}$$

(L$^{\tau}$-Plug)

$$\frac{\begin{array}{ccc}\mathsf{A} \equiv \mathsf{H}; \overline{\mathsf{F}}\,[\cdot] & & \mathsf{C} \equiv \Delta; \overline{\mathsf{F}'}; \bar{\mathsf{I}} \\ \vdash \mathsf{C}, \overline{\mathsf{F}} : \mathsf{whole} & \Delta \vdash \mathsf{H}_0 & \mathsf{main}(\mathsf{x} : \mathsf{UN}) \mapsto \mathsf{s}; \mathsf{return}; \in \overline{\mathsf{F}}\end{array}}{\mathsf{A}\,[\mathsf{C}] = \Delta; \mathsf{H} \cup \mathsf{H}_0; \overline{\mathsf{F}; \mathsf{F}'}; \bar{\mathsf{I}}}$$

(L$^{\tau}$-Whole)

$$\frac{\begin{array}{c}\mathsf{C} \equiv \Delta; \overline{\mathsf{F}'}; \bar{\mathsf{I}} \\ \texttt{names}(\overline{\mathsf{F}}) \cap \texttt{names}(\overline{\mathsf{F}'}) = \varnothing \\ \texttt{names}(\bar{\mathsf{I}}) \subseteq \texttt{names}(\overline{\mathsf{F}}) \cup \texttt{names}(\overline{\mathsf{F}'})\end{array}}{\vdash \mathsf{C}, \overline{\mathsf{F}} : \mathsf{whole}}$$

(L$^{\tau}$-Initial State)

$$\frac{\begin{array}{cc}\multicolumn{2}{c}{\mathsf{P} \equiv \Delta; \mathsf{H}; \overline{\mathsf{F}}; \bar{\mathsf{I}}} \\ \mathsf{C} \equiv \Delta; \overline{\mathsf{F}}; \bar{\mathsf{I}} & \mathsf{main}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \in \overline{\mathsf{F}}\end{array}}{\Omega_0\,(\mathsf{P}) = \mathsf{C}, \mathsf{H} \triangleright (\mathsf{s}[0 \,/\, \mathsf{x}])_{\mathsf{main}}}$$

$$\boxed{\Delta \vdash \mathsf{H}_0}$$

(L$^{\tau}$-Initial-heap)

$$\frac{\Delta \vdash \mathsf{H} \qquad \varnothing \vdash \mathsf{v} : \tau}{\Delta, \ell : \tau \vdash \mathsf{H}; \ell \mapsto \mathsf{v} : \tau}$$

$$\boxed{\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'}$$

$$\frac{\mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_{\mathsf{c}}) \qquad \mathsf{M}' = (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_{\mathsf{f}}) \qquad (\sigma_{\mathsf{c}}, \sigma_{\mathsf{f}}) \in \rightsquigarrow \qquad \vdash \mathsf{H} : \Delta}{\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'}\ (\mathsf{L}^{\tau}\text{-Monitor Step})$$

$$\frac{}{\mathsf{M}; \varnothing \rightsquigarrow \mathsf{M}}\ (\mathsf{L}^{\tau}\text{-Monitor Step Trace Base}) \qquad \frac{\mathsf{M}; \overline{\mathsf{H}} \rightsquigarrow \mathsf{M}'' \qquad \mathsf{M}''; \mathsf{H} \rightsquigarrow \mathsf{M}'}{\mathsf{M}; \overline{\mathsf{H}} \cdot \mathsf{H} \rightsquigarrow \mathsf{M}'}\ (\mathsf{L}^{\tau}\text{-Monitor Step Trace})$$

$$\frac{\mathsf{M}; \overline{\mathsf{H}} \rightsquigarrow \mathsf{M}' \qquad \mathtt{heaps}(\overline{\alpha}) = \overline{\mathsf{H}}}{\mathsf{M} \vdash \overline{\alpha}}\ (\mathsf{L}^{\tau}\text{-valid trace})$$

5.2.1 Component Semantics

$\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \mathsf{e}'$ Expression e reduces to e'.

$\mathsf{C}, \mathsf{H} \triangleright \pi \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \triangleright \pi$ Process π reduces to π' and evolves the rest accordingly.

$\mathsf{C}, \mathsf{H} \triangleright \Pi \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \triangleright \Pi'$ Soup Π reduce to Π' and evolve the rest accordingly.

$\Omega \xRightarrow{\overline{\alpha}} \Omega'$ Program state Ω steps to Ω' emitting trace $\overline{\alpha}$.

$$\boxed{\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \mathsf{e}'}$$

$$\frac{}{\mathsf{H} \triangleright \mathsf{v} \lhook\joinrel\twoheadrightarrow \mathsf{v}}\ (\mathsf{EL}^{\tau}\text{-val}) \qquad \frac{}{\mathsf{H} \triangleright \langle \mathsf{v}, \mathsf{v}' \rangle .1 \lhook\joinrel\twoheadrightarrow \mathsf{v}}\ (\mathsf{EL}^{\tau}\text{-p1}) \qquad \frac{}{\mathsf{H} \triangleright \langle \mathsf{v}, \mathsf{v}' \rangle .1 \lhook\joinrel\twoheadrightarrow \mathsf{v}'}\ (\mathsf{EL}^{\tau}\text{-p2})$$

$$\frac{n \oplus n' = n''}{\mathsf{H} \triangleright \mathsf{n} \oplus \mathsf{n}' \lhook\joinrel\twoheadrightarrow \mathsf{n}''}\ (\mathsf{EL}^{\tau}\text{-op}) \qquad \frac{n \otimes n' = b}{\mathsf{H} \triangleright \mathsf{n} \otimes \mathsf{n}' \lhook\joinrel\twoheadrightarrow \mathsf{b}}\ (\mathsf{EL}^{\tau}\text{-comp})$$

$$\frac{\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \ell \qquad \ell \mapsto \mathsf{v} : \tau \in \mathsf{H}}{\mathsf{H} \triangleright ! \ell \lhook\joinrel\twoheadrightarrow \mathsf{v}}\ (\mathsf{EL}^{\tau}\text{-dereference}) \qquad \frac{\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \mathsf{e}'}{\mathsf{H} \triangleright \mathsf{E}[\mathsf{e}] \lhook\joinrel\twoheadrightarrow \mathsf{E}[\mathsf{e}']}\ (\mathsf{EL}^{\tau}\text{-ctx})$$

$$\boxed{\mathsf{C}, \mathsf{H} \triangleright \pi \xrightarrow{\epsilon} \mathsf{C}', \mathsf{H}' \triangleright \pi'}$$

$$\frac{}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{skip}; \mathsf{s} \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}}\ (\mathsf{EL}^{\tau}\text{-sequence}) \qquad \frac{\mathsf{C}, \mathsf{H} \triangleright \mathsf{s} \xrightarrow{\lambda} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}'}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{s}; \mathsf{s}'' \xrightarrow{\lambda} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}'; \mathsf{s}}\ (\mathsf{EL}^{\tau}\text{-step})$$

$$\frac{\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \mathsf{true}}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s}' \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}}\ (\mathsf{EL}^{\tau}\text{-if-true})$$

$$\frac{\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \mathsf{false}}{\mathsf{C}, \mathsf{H} \triangleright \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s}' \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}'}\ (\mathsf{EL}^{\tau}\text{-if-false})$$

(EL$^\tau$-letin)

$$\frac{H \triangleright e \lhook\joinrel\twoheadrightarrow v}{C, H \triangleright \text{let } x : \tau = e \text{ in } s \xrightarrow{\epsilon} C, H \triangleright s[v \,/\, x]}$$

(EL$^\tau$-alloc)

$$\frac{\ell \notin \mathtt{dom}(H) \qquad H \triangleright e \lhook\joinrel\twoheadrightarrow v}{C, H \triangleright \text{let } x = \text{new}_\tau \; e \text{ in } s \xrightarrow{\epsilon} C, H; \ell \mapsto v : \tau \triangleright s[\ell \,/\, x]}$$

(EL$^\tau$-update)

$$\frac{\begin{array}{c} H = H_1; \ell \mapsto v' : \tau; H_2 \\ H' = H_1; \ell \mapsto v : \tau; H_2 \end{array}}{C, H \triangleright \ell := v \xrightarrow{\epsilon} C, H' \triangleright \text{skip}}$$

(EL$^\tau$-endorse)

$$\frac{H \triangleright e \lhook\joinrel\twoheadrightarrow v \qquad \Delta, \varnothing \vdash v : \varphi \qquad \Delta = \{\ell : \tau \mid \ell \mapsto v : \tau \in H\}}{C, H \triangleright \text{endorse } x = e \text{ as } \varphi \text{ in } s \xrightarrow{\epsilon} C, H \triangleright s[v \,/\, x]}$$

(EL$^\tau$-call-internal)

$$\frac{\begin{array}{cc} \overline{C}.\mathtt{intfs} \vdash f, f' : \text{internal} & \overline{f'} = \overline{f''}; f' \\ f(x : \tau) : \tau' \mapsto s; \text{return}; \in C.\mathtt{funs} & H \triangleright e \lhook\joinrel\twoheadrightarrow v \end{array}}{C, H \triangleright (\text{call } f \; e)_{\overline{f'}} \xrightarrow{\epsilon} C, H \triangleright (s; \text{return};[v \,/\, x])_{\overline{f'};f}}$$

(EL$^\tau$-callback)

$$\frac{\begin{array}{cc} \overline{f'} = \overline{f''}; f' & f(x : \tau) : \tau' \mapsto s; \text{return}; \in \overline{F} \\ \overline{C}.\mathtt{intfs} \vdash f', f : \text{out} & H \triangleright e \lhook\joinrel\twoheadrightarrow v \end{array}}{C, H \triangleright (\text{call } f \; e)_{\overline{f'}} \xrightarrow{\mathtt{call\ f\ v!}} C, H \triangleright (s; \text{return};[v \,/\, x])_{\overline{f'};f}}$$

(EL$^\tau$-call)

$$\frac{\begin{array}{cc} \overline{f'} = \overline{f''}; f' & f(x : \tau) : \tau' \mapsto s; \text{return}; \in C.\mathtt{funs} \\ \overline{C}.\mathtt{intfs} \vdash f', f : \text{in} & H \triangleright e \lhook\joinrel\twoheadrightarrow v \end{array}}{C, H \triangleright (\text{call } f \; e)_{\overline{f'}} \xrightarrow{\mathtt{call\ f\ v?}} C, H \triangleright (s; \text{return};[v \,/\, x])_{\overline{f'};f}}$$

(EL$^\tau$-ret-internal)

$$\frac{\overline{C}.\mathtt{intfs} \vdash f, f' : \text{internal} \qquad \overline{f'} = \overline{f''}; f'}{C, H \triangleright (\text{return};)_{\overline{f'};f} \xrightarrow{\epsilon} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

(EL$^\tau$-retback)

$$\frac{\overline{C}.\mathtt{intfs} \vdash f, f' : \text{in} \qquad \overline{f'} = \overline{f''}; f'}{C, H \triangleright (\text{return};)_{\overline{f'};f} \xrightarrow{\mathtt{ret\ ?}} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

(EL$^\tau$-return)

$$\frac{\overline{C}.\mathtt{intfs} \vdash f, f' : \text{out} \qquad \overline{f'} = \overline{f''}; f' \qquad H \triangleright e \lhook\joinrel\twoheadrightarrow v}{C, H \triangleright (\text{return};)_{\overline{f'};f} \xrightarrow{\mathtt{ret\ !}} C, H \triangleright (\text{skip})_{\overline{f'}}}$$

$$\boxed{C, H \triangleright \Pi \xrightarrow{\lambda} C', H' \triangleright \Pi'}$$

$$\frac{\begin{array}{c}\Pi = \Pi_1 \parallel (\mathsf{s})_{\overline{\mathsf{f}}} \parallel \Pi_2 \\ \Pi' = \Pi_1 \parallel (\mathsf{s}')_{\overline{\mathsf{f}'}} \parallel \Pi_2 \\ \mathsf{C}, \mathsf{H} \rhd (\mathsf{s})_{\overline{\mathsf{f}}} \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \rhd (\mathsf{s}')_{\overline{\mathsf{f}'}}\end{array}}{\mathsf{C}, \mathsf{H} \rhd \Pi \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \rhd \Pi'}\ (\mathsf{EL}^{\tau}\text{-par})$$

$$\frac{\begin{array}{c}\Pi = \Pi_1 \parallel (\mathsf{s})_{\overline{\mathsf{f}}} \parallel \Pi_2 \\ \mathsf{C}, \mathsf{H} \rhd (\mathsf{s})_{\overline{\mathsf{f}}} \xrightarrow{\epsilon} \mathsf{fail}\end{array}}{\mathsf{C}, \mathsf{H} \rhd \Pi \xrightarrow{\epsilon} \mathsf{fail}}\ (\mathsf{EL}^{\tau}\text{-fail})$$

$$\frac{\begin{array}{c}\Pi = \Pi_1 \parallel ((\parallel \mathsf{s})\,;\mathsf{s}')_{\overline{\mathsf{f}}} \parallel \Pi_2 \\ \Pi' = \Pi_1 \parallel (\mathsf{skip};\mathsf{s}')_{\overline{\mathsf{f}}} \parallel \Pi_2 \parallel (\mathsf{s})_{\varnothing}\end{array}}{\mathsf{C}, \mathsf{H} \rhd \Pi \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H} \rhd \Pi'}\ (\mathsf{EL}^{\tau}\text{-fork})$$

$$\boxed{\Omega \overset{\overline{\alpha}}{\Longrightarrow} \Omega'}$$

$$\frac{\Omega \xrightarrow{\alpha} \Omega'}{\Omega \overset{\alpha}{\Longrightarrow} \Omega'}\ (\mathsf{EL}^{\tau}\text{-single}) \qquad \frac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Longrightarrow \Omega'}\ (\mathsf{EL}^{\tau}\text{-silent}) \qquad \frac{\begin{array}{c}\Omega \overset{\overline{\alpha}}{\Longrightarrow} \Omega'' \\ \Omega'' \overset{\overline{\alpha'}}{\Longrightarrow} \Omega'\end{array}}{\Omega \overset{\overline{\alpha}\cdot\overline{\alpha'}}{\Longrightarrow} \Omega'}\ (\mathsf{EL}^{\tau}\text{-trans})$$

6 $\mathbf{L}^{\pi}$: Extending $\mathbf{L^P}$ with Concurrency and Informed Monitors

6.1 Syntax

This extends the syntax of Section 2.1 with concurrency and a memory allocation instruction that atomically hides the new location.

$$\begin{aligned}
\textit{Whole Programs } \mathbf{P} &::= \mathbf{H_0}; \overline{\mathbf{F}}; \overline{\mathbf{I}} \\
\textit{Components } \mathbf{C} &::= \mathbf{H_0}; \overline{\mathbf{F}}; \overline{\mathbf{I}} \\
\textit{Statements } \mathbf{s} &::= \cdots \mid (\parallel \mathbf{s}) \mid \mathbf{destruct\ x = e\ as\ B\ in\ s\ or\ s} \\
&\quad \mid \mathbf{let\ x = newhide\ e\ in\ s} \\
\textit{Patterns } \mathbf{B} &::= \mathbf{nat} \mid \mathbf{pair} \\
\textit{Monitors } \mathbf{M} &::= (\{\sigma\}, \rightsquigarrow, \sigma_0, \mathbf{H_0}, \sigma_c) \\
\textit{Single Process } \pi &::= (\mathbf{s})_{\overline{\mathbf{f}}} \\
\textit{Processes } \mathbf{\Pi} &::= \varnothing \mid \mathbf{\Pi} \parallel \pi \\
\textit{Prog. States } \mathbf{\Omega} &::= \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi}
\end{aligned}$$

6.2 Dynamic Semantics

Following is the definition of the $\texttt{mon-care}(\,\cdot\,)$ function for $\mathbf{L}^{\pi}$.

$\boxed{\texttt{mon-care}(\,\cdot\,)}$

$$\frac{\mathbf{H}' = \{\mathbf{n} \mapsto \mathbf{v} : \eta \mid \mathbf{n} \in \texttt{dom}(\mathbf{H_0}) \text{ and } \mathbf{n} \mapsto \mathbf{v} : \eta \in \mathbf{H}\}}{\texttt{mon-care}(\mathbf{H}, \mathbf{H_0}) = \mathbf{H}'}\ (\mathbf{L}^{\pi}\text{-Monitor-related heap})$$

$\boxed{\text{Helpers}}$

$$\frac{\begin{array}{cc} \mathbf{A} \equiv \overline{\mathbf{F}}\,[\cdot] & \mathbf{C} \equiv \mathbf{H_0}; \overline{\mathbf{F'}}; \overline{\mathbf{I}} \\ \vdash \mathbf{C}, \overline{\mathbf{F}} : \mathbf{whole} & \mathbf{main(x) \mapsto return;s} \in \overline{\mathbf{F}} \\ \mathbf{C} \vdash_{\mathbf{att}} \mathbf{A} & \forall \mathbf{n} \mapsto \mathbf{v} : \mathbf{k} \in \mathbf{H_0}, \mathbf{k} \in \mathbf{H_0} \end{array}}{\mathbf{A}\,[\mathbf{C}] = \mathbf{H_0}; \overline{\mathbf{F}; \mathbf{F'}}; \overline{\mathbf{I}}}\ (\mathbf{L}^{\pi}\text{-Plug})$$

$$\frac{\mathbf{P} \equiv \mathbf{H_0}; \overline{\mathbf{F}}; \overline{\mathbf{I}} \qquad \mathbf{main(x) \mapsto s; return;} \in \overline{\mathbf{F}}}{\mathbf{\Omega_0}\,(\mathbf{P}) = \mathbf{P}, \mathbf{H_0} \triangleright (\mathbf{s}[\mathbf{0} \,/\, \mathbf{x}])_{\mathbf{main}}}\ (\mathbf{L}^{\pi}\text{-Initial State})$$

$\boxed{\mathbf{M}; \mathbf{H} \rightsquigarrow \mathbf{M}'}$

$$\dfrac{\begin{array}{c}\mathbf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{H_0}, \sigma_{\mathbf{c}}) \qquad \mathbf{M'} = (\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{H_0}, \sigma_{\mathbf{f}}) \\ (\mathbf{s_c}, \texttt{mon-care}(\mathbf{H}, \mathbf{H_0}), \mathbf{s_f}) \in \rightsquigarrow\end{array}}{\mathbf{M}; \mathbf{H} \rightsquigarrow \mathbf{M'}}\ (\mathbf{L}^{\pi}\text{-Monitor Step})$$

$$\dfrac{}{\mathbf{M}; \varnothing \rightsquigarrow \mathbf{M}}\ (\mathbf{L}^{\pi}\text{-Monitor Step Trace Base}) \qquad \dfrac{\mathbf{M}; \overline{\mathbf{H}} \rightsquigarrow \mathbf{M''} \qquad \mathbf{M''}; \mathbf{H} \rightsquigarrow \mathbf{M'}}{\mathbf{M}; \overline{\mathbf{H}} \cdot \mathbf{H} \rightsquigarrow \mathbf{M'}}\ (\mathbf{L}^{\pi}\text{-Monitor Step Trace})$$

$$\dfrac{\mathbf{M}; \overline{\mathbf{H}} \rightsquigarrow \mathbf{M'}}{\mathbf{M} \vdash \overline{\texttt{mon}\ \mathbf{H}}}\ (\mathbf{L}^{\pi}\text{-valid trace})$$

6.2.1 Component Semantics

$\mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi} \xrightarrow{\epsilon} \mathbf{C'}, \mathbf{H'} \triangleright \mathbf{\Pi'}$ Processes $\mathbf{\Pi}$ reduce to $\mathbf{\Pi'}$ and evolve the rest accordingly.

$$\boxed{\mathbf{C}, \mathbf{H} \triangleright \mathbf{s} \xrightarrow{\epsilon} \mathbf{C'}, \mathbf{H'} \triangleright \mathbf{s'}}$$

$$\dfrac{\mathbf{H} \triangleright \mathbf{e} \lhook\joinrel\twoheadrightarrow \mathbf{n}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{destruct\ x = e\ as\ nat\ in\ s\ or\ s'} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}[\mathbf{n} \,/\, \mathbf{x}]}\ (\mathsf{E}\mathbf{L}^{\pi}\text{-destruct-nat})$$

$$\dfrac{\mathbf{H} \triangleright \mathbf{e} \lhook\joinrel\twoheadrightarrow \langle \mathbf{v}, \mathbf{v'} \rangle}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{destruct\ x = e\ as\ pair\ in\ s\ or\ s'} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s}[\langle \mathbf{v}, \mathbf{v'} \rangle \,/\, \mathbf{x}]}\ (\mathsf{E}\mathbf{L}^{\pi}\text{-destruct-pair})$$

$$\dfrac{\text{otherwise}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{destruct\ x = e\ as\ B\ in\ s\ or\ s'} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H} \triangleright \mathbf{s'}}\ (\mathsf{E}\mathbf{L}^{\pi}\text{-destruct-not})$$

$$\dfrac{\mathbf{H} = \mathbf{H_1}; \mathbf{n} \mapsto (\mathbf{v}, \eta) \qquad \mathbf{H} \triangleright \mathbf{e} \lhook\joinrel\twoheadrightarrow \mathbf{v} \qquad \mathbf{k} \notin \texttt{dom}(\mathbf{H})}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x = newhide\ e\ in\ s} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H}; \mathbf{n+1} \mapsto \mathbf{v} : \mathbf{k}; \mathbf{k} \triangleright \mathbf{s}[\langle \mathbf{n+1}, \mathbf{k} \rangle \,/\, \mathbf{x}]}\ (\mathsf{E}\mathbf{L}^{\pi}\text{-new})$$

$$\boxed{\mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi} \lhook\joinrel\twoheadrightarrow \mathbf{C'}, \mathbf{H'} \triangleright \mathbf{\Pi'}}$$

$$\dfrac{\begin{array}{c}\mathbf{\Pi} = \mathbf{\Pi_1} \parallel (\mathbf{s})_{\overline{\mathbf{f}}} \parallel \mathbf{\Pi_2} \\ \mathbf{\Pi'} = \mathbf{\Pi_1} \parallel (\mathbf{s'})_{\overline{\mathbf{f'}}} \parallel \mathbf{\Pi_2} \\ \mathbf{C}, \mathbf{H} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}} \lhook\joinrel\twoheadrightarrow \mathbf{C'}, \mathbf{H'} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}\end{array}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi} \lhook\joinrel\twoheadrightarrow \mathbf{C'}, \mathbf{H'} \triangleright \mathbf{\Pi'}}\ (\mathsf{E}\mathbf{L}^{\pi}\text{-par}) \qquad \dfrac{\begin{array}{c}\mathbf{\Pi} = \mathbf{\Pi_1} \parallel ((\parallel \mathbf{s}))_{\overline{\mathbf{f}}} \parallel \mathbf{\Pi_2} \\ \mathbf{\Pi'} = \mathbf{\Pi_1} \parallel (\mathbf{0})_{\overline{\mathbf{f}}} \parallel \mathbf{\Pi_2} \parallel (\mathbf{s})_{\varnothing}\end{array}}{\mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi} \lhook\joinrel\twoheadrightarrow \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi'}}\ (\mathsf{E}\mathbf{L}^{\pi}\text{-fork})$$

7 Extended Language Properties and Necessities

7.1 Monitor Agreement for L^{τ} and $\mathbf{L}^{\pi}$

Definition 16 (L^{τ}: $\mathsf{M} \frown \mathsf{C}$)**.**

$$(\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_{\mathsf{c}}) \frown (\Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}})$$

A monitor and a component agree if they focus on the same set of locations Δ.

Definition 17 ($\mathbf{L}^{\pi}$: $\mathbf{M} \frown \mathbf{C}$)**.**

$$(\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{H_0}, \sigma_{\mathbf{c}}) \frown (\mathbf{H_0}; \overline{\mathbf{F}}; \overline{\mathbf{I}})$$

A monitor and a component agree if they focus on the same set of locations, protected with the same capabilities $\mathbf{H_0}$

7.2 Properties of L^{τ}

Definition 18 (L^{τ} Semantics Attacker)**.**

$$\mathsf{C} \vdash_{\mathsf{attacker}} \mathsf{A} \overset{\mathsf{def}}{=} \begin{cases} \forall \ell \in \mathtt{dom}(\mathsf{C}.\Delta), \ell \notin \mathtt{locs}(\mathsf{A}) \\ \text{no } \mathsf{let\ x = new_{\tau}\ e\ in\ s} \text{ in } \mathsf{A} \text{ such that } \tau \neq \mathsf{UN} \end{cases}$$

This semantic definition of an attacker is captured by typing below, which allows for simpler reasoning.

Definition 19 (L^{τ} Attacker)**.**

$$\mathsf{C} \vdash_{\mathsf{att}} \mathsf{A} \overset{\mathsf{def}}{=} \mathsf{C} = \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}, \Delta \vdash_{\mathsf{UN}} \mathsf{A}$$

$$\mathsf{C} \vdash_{\mathsf{att}} \pi \overset{\mathsf{def}}{=} \pi = (\mathsf{s})_{\overline{\mathsf{f}};\mathsf{f}} \text{ and } \mathsf{f} \in \mathsf{C}.\mathtt{itfs}$$

$$\mathsf{C} \vdash_{\mathsf{att}} \Pi \longrightarrow \Pi' \overset{\mathsf{def}}{=} \Pi = \Pi_1 \parallel \pi \parallel \Pi_2 \text{ and } \Pi' = \Pi_1 \parallel \pi' \parallel \Pi_2$$
$$\text{and } \mathsf{C} \vdash_{\mathsf{att}} \pi \text{ and } \mathsf{C} \vdash_{\mathsf{att}} \pi'$$

The two notions of attackers coincide.

Lemma 4 (Semantics and typed attackers coincide)**.**

$$\mathsf{C} \vdash_{\mathsf{attacker}} \mathsf{A} \iff (\mathsf{C} \vdash_{\mathsf{att}} \mathsf{A})$$

Theorem 5 (Typability Implies Robust Safety in L^{τ})**.**

$$\begin{gathered} \forall \mathsf{C}, \mathsf{M} \\ \text{if } \vdash \mathsf{C} : \mathsf{UN} \\ \mathsf{C} \frown \mathsf{M} \\ \text{then } \mathsf{M} \vdash \mathsf{C} : \mathsf{rs} \end{gathered}$$

7.3 Properties of $\mathbf{L}^{\pi}$

Definition 20 ($\mathbf{L}^{\pi}$ Attacker)**.**

$$\mathbf{C} \vdash_{\mathbf{att}} \mathbf{A} \overset{\mathsf{def}}{=} \mathbf{C} = \mathbf{H_0}; \overline{\mathbf{F}}; \bar{\mathbf{I}}, \forall \mathbf{k} \in \mathbf{H_0}.\mathbf{k} \notin \mathtt{fv}(\mathbf{A})$$

$$\mathbf{C} \vdash_{\mathbf{att}} \pi \overset{\mathsf{def}}{=} \pi = (\mathbf{s})_{\overline{\mathbf{f}};\mathbf{f}} \text{ and } \mathbf{f} \in \mathbf{C}.\mathtt{itfs}$$

$$\begin{aligned} \mathbf{C} \vdash_{\mathbf{att}} \mathbf{\Pi} \longrightarrow \mathbf{\Pi}' \overset{\mathsf{def}}{=} & \ \mathbf{\Pi} = \mathbf{\Pi_1} \parallel \pi \parallel \mathbf{\Pi_2} \text{ and } \mathbf{\Pi}' = \mathbf{\Pi_1} \parallel \pi' \parallel \mathbf{\Pi_2} \\ & \text{and } \mathbf{C} \vdash_{\mathbf{att}} \pi \text{ and } \mathbf{C} \vdash_{\mathbf{att}} \pi' \end{aligned}$$

8 Compiler from L^τ to $\mathbf{L}^\pi$

8.1 Assumed Relation between L^τ and $\mathbf{L}^\pi$ Elements

We can scale the $\approx_\beta$ relation to monitors, heaps, actions and processes as follows.

$$\boxed{\mathsf{M} \approx \mathbf{M}}$$

$$\frac{\begin{array}{c}\mathbf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \mathbf{H_0}, \sigma_c) \\ \forall \sigma \in \{\sigma\}, \texttt{mon-care}(\mathsf{H}; \Delta) \approx_\beta \texttt{mon-care}(\mathbf{H}, \mathbf{H_0}). \\ \text{if } \vdash \mathsf{H} : \Delta \text{ then } \exists \sigma'.(\sigma, \texttt{mon-care}(\mathbf{H}, \mathbf{H_0}), \sigma') \in \rightsquigarrow \end{array}}{\beta, \Delta \vdash \mathbf{M}} \;\text{(Ok Mon)}$$

$$\frac{\begin{array}{cc}\mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c) & \mathbf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \mathbf{H_0}, \sigma_c) \\ \beta_0, \Delta \vdash \mathbf{M} & \beta_0 = (\texttt{dom}(\Delta), \texttt{dom}(\mathbf{H_0}), \mathbf{H_0}.\eta) \end{array}}{\mathsf{M} \approx \mathbf{M}} \;\text{(Monitor relation)}$$

$$\boxed{\Delta \vdash_\beta \mathbf{H_0} \quad \Delta, \mathbf{H} \vdash \mathbf{v} : \tau}$$

$$\frac{\begin{array}{cc}\Delta \vdash \mathbf{H} & \Delta, \mathbf{H} \vdash_\beta \mathbf{v} \colon \tau \\ \multicolumn{2}{c}{\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle}\end{array}}{\Delta, \ell : \tau \vdash_\beta \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \mathbf{k}} \;\text{(Initial-heap)}$$

$$\frac{\begin{array}{c}(\tau \equiv \mathsf{Bool} \land \mathbf{v} \equiv \mathbf{0}) \qquad \lor \qquad (\tau \equiv \mathsf{Nat} \land \mathbf{v} \equiv \mathbf{0}) \qquad \lor \\ (\tau \equiv \mathsf{Ref}\ \tau \land \mathbf{v} \equiv \mathbf{n'} \land \mathbf{n'} \mapsto \mathbf{v'} : \mathbf{k'} \in \mathbf{H} \land \ell' \approx_\beta \langle \mathbf{n'}, \mathbf{k'} \rangle \land \ell : \tau \in \Delta, \Delta, \mathbf{H} \vdash \mathbf{v'} \colon \tau) \qquad \lor \\ (\tau \equiv \tau_1 \times \tau_2 \land \mathbf{v} \equiv \langle \mathbf{v_1}, \mathbf{v_2} \rangle \land \Delta, \mathbf{H} \vdash \mathbf{v_1} \colon \tau_1 \land \Delta, \mathbf{H} \vdash \mathbf{v_2} \colon \tau_2)\end{array}}{\Delta, \mathbf{H} \vdash_\beta \mathbf{v} \colon \tau} \;\text{(Initial-value)}$$

$$\boxed{\Pi \approx_\beta \mathbf{\Pi}}$$

$$\frac{\bar{\mathsf{f}} \approx \bar{\mathbf{f}}}{(\mathsf{skip})_{\bar{\mathsf{f}}} \approx_\beta (\mathbf{skip})_{\bar{\mathbf{f}}}} \;\text{(Single process relation)} \qquad \frac{\Pi \approx_\beta \mathbf{\Pi} \qquad \pi \approx_\beta \pi}{\Pi \parallel \pi \approx_\beta \mathbf{\Pi} \parallel \pi} \;\text{(Process relation)}$$

8.2 Compiler Definition

Definition 21 (Compiler L^τ to $\mathbf{L}^\pi$)**.** $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} : \mathsf{C} \to \mathbf{C}$

Given that $\mathsf{C} = \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}$ if $\vdash \mathsf{C} : \mathsf{UN}$ then $\llbracket \mathsf{C} \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$ is defined as follows:

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-component})\\ \mathsf{C} \equiv \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}\\ \mathsf{C} \vdash \overline{\mathsf{F}} : \mathsf{UN}\\ \mathtt{names}(\overline{\mathsf{F}}) \cap \mathtt{names}(\overline{\mathsf{I}}) = \varnothing\\ \Delta \vdash \mathsf{ok}\end{array}}{\vdash \mathsf{C} : \mathsf{UN}} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{H_0}; \llbracket \overline{\mathsf{F}} \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}; \llbracket \overline{\mathsf{I}} \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \qquad \text{if } \Delta \vdash_{\beta_0} \mathbf{H_0}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Component})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-function})\\ \mathsf{F} \equiv \mathsf{f}(\mathsf{x} : \mathsf{UN}) \mapsto \mathsf{s}; \mathsf{return};\\ \mathsf{C}, \Delta; \mathsf{x} : \mathsf{UN} \vdash \mathsf{s}\\ \forall \mathsf{f} \in \mathtt{fn}(s), \mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{funs})\\ \forall \mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{intfs})\end{array}}{\mathsf{C} \vdash \mathsf{F} : \mathsf{UN}} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{f(x)} \mapsto \llbracket \mathsf{C}; \Delta; \mathsf{x} : \mathsf{UN} \vdash \mathsf{s} \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}; \mathbf{return};$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Function})$$

$$\llbracket \mathsf{f} \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{f} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Interfaces})$$

Expressions

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-true})\\ \Delta, \Gamma \vdash \diamond\end{array}}{\Delta, \Gamma \vdash \mathsf{true} : \mathsf{Bool}} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{0} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-True})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-false})\\ \Delta, \Gamma \vdash \diamond\end{array}}{\Delta, \Gamma \vdash \mathsf{false} : \mathsf{Bool}} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{1} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-False})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-nat})\\ \Delta, \Gamma \vdash \diamond\end{array}}{\Delta, \Gamma \vdash \mathsf{n} : \mathsf{Nat}} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{n} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Nat})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-var})\\ \mathsf{x} : \tau \in \Gamma\end{array}}{\Delta, \Gamma \vdash \mathsf{x} : \tau} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{x} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Var})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-loc})\\ \ell : \tau \in \Delta\end{array}}{\Delta, \Gamma \vdash \ell : \tau} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \langle \mathbf{n}, \mathbf{v} \rangle \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Loc})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^\tau\text{-pair})\\ \Delta, \Gamma \vdash \mathsf{e}_1 : \tau\\ \Delta, \Gamma \vdash \mathsf{e}_2 : \tau'\end{array}}{\Delta, \Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \tau \times \tau'} \right\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \left\langle \llbracket \Delta, \Gamma \vdash \mathsf{e}_1 : \tau \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}, \llbracket \Delta, \Gamma \vdash \mathsf{e}_2 : \tau' \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \right\rangle$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\text{-Pair})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-proj-1})\\ \dfrac{\mathsf{\Delta,\Gamma \vdash e : \tau \times \tau'}}{\mathsf{\Delta,\Gamma \vdash e.1 : \tau}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{\Delta,\Gamma \vdash e : \tau \times \tau'} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}.\mathbf{1} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-P1})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-proj-2})\\ \dfrac{\mathsf{\Delta,\Gamma \vdash e : \tau \times \tau'}}{\mathsf{\Delta,\Gamma \vdash e.2 : \tau'}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{\Delta,\Gamma \vdash e : \tau \times \tau'} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}.\mathbf{2} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-P2})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-dereference})\\ \dfrac{\mathsf{\Delta,\Gamma \vdash e : Ref\ \tau}}{\mathsf{\Delta,\Gamma \vdash\ !e : \tau}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \mathbf{!}\llbracket \mathsf{\Delta,\Gamma \vdash e : Ref\ \tau} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}.\mathbf{1}\ \mathbf{with}\ \llbracket \mathsf{\Delta,\Gamma \vdash e : Ref\ \tau} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}.\mathbf{2}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Deref})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-op})\\ \dfrac{\begin{array}{c}\mathsf{\Delta,\Gamma \vdash e : Nat}\\ \mathsf{\Delta,\Gamma \vdash e' : Nat}\end{array}}{\mathsf{\Delta,\Gamma \vdash e \oplus e' : Nat}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{\Delta,\Gamma \vdash e : Nat} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \oplus \llbracket \mathsf{\Delta,\Gamma \vdash e' : Nat} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-op})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-cmp})\\ \dfrac{\begin{array}{c}\mathsf{\Delta,\Gamma \vdash e : Nat}\\ \mathsf{\Delta,\Gamma \vdash e' : Nat}\end{array}}{\mathsf{\Delta,\Gamma \vdash e \otimes e' : Bool}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{\Delta,\Gamma \vdash e : Nat} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \otimes \llbracket \mathsf{\Delta,\Gamma \vdash e' : Nat} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-cmp})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-coercion})\\ \dfrac{\mathsf{\Delta,\Gamma \vdash e : \tau} \qquad \mathsf{\tau \vdash \circ}}{\mathsf{\Delta,\Gamma \vdash e : UN}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{\Delta,\Gamma \vdash e : \tau} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Coerce})$$

Statements

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-skip})\\ \dfrac{}{\mathsf{C,\Delta,\Gamma \vdash skip}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \mathbf{skip} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Skip})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-new})\\ \dfrac{\begin{array}{c}\mathsf{C,\Delta,\Gamma \vdash e : \tau}\\ \mathsf{C,\Delta,\Gamma; x : Ref\ \tau \vdash s}\end{array}}{\mathsf{C,\Delta,\Gamma \vdash let\ x = new_{\tau}\ e\ in\ s}}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{cases} \mathbf{let\ xo\ = new}\ \llbracket \mathsf{\Delta,\Gamma \vdash e : \tau} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in\ let\ x} = \langle \mathbf{xo}, \mathbf{0} \rangle \\ \quad\ \mathbf{in}\ \llbracket \mathsf{C,\Delta,\Gamma; x : Ref\ \tau \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \text{if } \tau = \mathsf{UN} \\ \\ \mathbf{let\ x\ = newhide}\ \llbracket \mathsf{\Delta,\Gamma \vdash e : \tau} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in}\ \llbracket \mathsf{C,\Delta,\Gamma; x : Ref\ \tau \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \text{else} \end{cases}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-New})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-function-call})\\ ((\mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{funs}))\\ \vee(\mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{intfs})))\\ \Delta,\Gamma \vdash \mathsf{e} : \mathsf{UN}\end{array}}{\Delta,\Gamma \vdash \mathsf{call\ f\ e}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \textbf{call f } \llbracket \Delta,\Gamma \vdash \mathsf{e} : \mathsf{UN} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-call})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-if})\\ \Delta,\Gamma \vdash \mathsf{e} : \mathtt{Bool}\\ \mathsf{C},\Delta,\Gamma \vdash \mathsf{s_t}\\ \mathsf{C},\Delta,\Gamma \vdash \mathsf{s_e}\end{array}}{\mathsf{C},\Delta,\Gamma \vdash \mathsf{if\ e\ then\ s_t\ else\ s_e}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{array}{l}\textbf{ifz } \llbracket \Delta,\Gamma \vdash \mathsf{e} : \mathtt{Bool} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \ \textbf{then } \llbracket \mathsf{C},\Delta,\Gamma \vdash \mathsf{s_t} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \ \textbf{else } \llbracket \mathsf{C},\Delta,\Gamma \vdash \mathsf{s_e} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\end{array}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-If})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-sequence})\\ \mathsf{C},\Delta,\Gamma \vdash \mathsf{s_u}\\ \mathsf{C},\Delta,\Gamma \vdash \mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma \vdash \mathsf{s_u};\mathsf{s}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{C},\Delta,\Gamma \vdash \mathsf{s_u} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}};\llbracket \mathsf{C},\Delta,\Gamma;\Gamma' \vdash \mathsf{s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Seq})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-letin})\\ \Delta,\Gamma \vdash \mathsf{e} : \tau\\ \mathsf{C},\Delta,\Gamma;\mathsf{x} : \tau \vdash \mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma \vdash \mathsf{let\ x} : \tau = \mathsf{e\ in\ s}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{array}{l}\textbf{let x}{=}\llbracket \Delta,\Gamma \vdash \mathsf{e} : \tau \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \ \textbf{in } \llbracket \mathsf{C},\Delta,\Gamma;\mathsf{x} : \tau \vdash \mathsf{s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\end{array}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Letin})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-assign})\\ \Delta,\Gamma \vdash \mathsf{x} : \mathsf{Ref}\ \tau\\ \Delta,\Gamma \vdash \mathsf{e} : \tau\end{array}}{\mathsf{C},\Delta,\Gamma \vdash \mathsf{x} := \mathsf{e}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{array}{l}\textbf{let x1} = \textbf{x.1}\\ \ \textbf{in let x2} = \textbf{x.2}\\ \quad \textbf{in x1} := \llbracket \Delta,\Gamma \vdash \mathsf{e} : \tau \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \textbf{ with x2}\end{array}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Assign})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-fork})\\ \mathsf{C},\Delta,\Gamma \vdash \mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma \vdash (\parallel \mathsf{s})} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \left(\parallel \llbracket \mathsf{C},\Delta,\Gamma \vdash \mathsf{s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\right)$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Fork})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-process})\\ \mathsf{C},\Delta,\Gamma \vdash \mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma \vdash (\mathsf{s})_{\overline{\mathsf{f}}}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \left(\llbracket \mathsf{C},\Delta,\Gamma \vdash \mathsf{s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\right)_{\llbracket \overline{\mathsf{f}} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Proc})$$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-soup})\\ \mathsf{C},\Delta,\Gamma \vdash \pi\\ \mathsf{C},\Delta,\Gamma \vdash \Pi\end{array}}{\mathsf{C},\Delta,\Gamma \vdash \pi \parallel \Pi} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \llbracket \mathsf{C},\Delta,\Gamma \vdash \pi \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \parallel \llbracket \mathsf{C},\Delta,\Gamma \vdash \Pi \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Soup})$$

$$\left\llbracket \begin{array}{c}(\mathsf{TL}^{\tau}\text{-endorse})\\ \mathsf{\Delta,\Gamma \vdash e : UN}\\ \mathsf{C,\Delta,\Gamma;(x:\varphi) \vdash s}\\ \hline \mathsf{C,\Delta,\Gamma \vdash endorse\ x = e\ as\ \varphi\ in\ s}\end{array} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{cases} \begin{array}{l}\mathbf{destruct\ x} \ = \llbracket \mathsf{\Delta,\Gamma \vdash e : UN} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\ \mathbf{as\ nat\ in}\\ \ \mathbf{ifz\ x\ then}\\ \quad \llbracket \mathsf{C,\Delta,\Gamma;(x:\varphi) \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \quad \mathbf{else\ ifz\ x - 1\ then}\\ \qquad \llbracket \mathsf{C,\Delta,\Gamma;(x:\varphi) \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \qquad \mathbf{else\ wrong}\\ \ \mathbf{or\ wrong}\\ \ \text{if } \varphi = \mathsf{Bool}\end{array} \\ \\ \begin{array}{l}\mathbf{destruct\ x} \ = \llbracket \mathsf{\Delta,\Gamma \vdash e : UN} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\ \mathbf{as\ nat\ in}\\ \ \llbracket \mathsf{C,\Delta,\Gamma;(x:\varphi) \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \ \mathbf{or\ wrong}\\ \ \text{if } \varphi = \mathsf{Nat}\end{array} \\ \\ \begin{array}{l}\mathbf{destruct\ x} \ = \llbracket \mathsf{\Delta,\Gamma \vdash e : UN} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\ \mathbf{as\ pair\ in}\\ \ \llbracket \mathsf{C,\Delta,\Gamma;(x:\varphi) \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \ \mathbf{or\ wrong}\\ \ \text{if } \varphi = \mathsf{UN \times UN}\end{array} \\ \\ \begin{array}{l}\mathbf{destruct\ x} \ = \llbracket \mathsf{\Delta,\Gamma \vdash e : UN} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\ \mathbf{as\ pair\ in}\\ \ \mathbf{!x.1\ with\ x.2;}\\ \quad \llbracket \mathsf{C,\Delta,\Gamma;(x:\varphi) \vdash s} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \ \mathbf{or\ wrong}\\ \ \text{if } \varphi = \mathsf{Ref\ UN}\end{array} \end{cases}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-Endorse})$$

We write **wrong** as a shortcut for a failign expression like $\mathbf{3 + true}$.

The remark about optimisation for $\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$ in Section 4 is also valid for the Rule ($\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-Deref) case above. As expressions are executed atomically, we are sure that albeit inefficient, dereferencing will correctly succeed.

We can add reference to superficial types and check this dynamically in the source, as we have the heap there. But how do we check this in the target? We only assume that reference must be passed as a pair: location- key from the attacker. Thus the last case of Rule ($\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-Endorse), where we check that we can access the location, otherwise we'd get stuck.

NonAtomic Implementation of New-Hide We can also implement Rule ($\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-New) using non-atomic instructions are defined in Rule ($\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-New-nonat) below.

$$\left\llbracket \dfrac{\overset{(\mathsf{TL}^{\tau}\text{-new})}{\mathsf{\Delta,\Gamma\vdash e:\tau}} \qquad \mathsf{C,\Delta,\Gamma;x:Ref\ \tau\vdash s}}{\mathsf{C,\Delta,\Gamma\vdash let\ x=new_{\tau}\ e\ in\ s}} \right\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} = \begin{cases} \begin{array}{l} \mathbf{let\ xo\ = new}\ \llbracket \mathsf{\Delta,\Gamma\vdash e:\tau}\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\ \quad \mathbf{in\ let\ x = \langle xo, 0\rangle} \\ \quad\ \mathbf{in}\ \llbracket \mathsf{C,\Delta,\Gamma;x:Ref\ \tau\vdash s}\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \end{array} & \text{if } \tau = \mathsf{UN} \\ \\ \begin{array}{l} \mathbf{let\ x_t\ = new\ 0\ in} \\ \ \mathbf{let\ x_k\ = hide\ x_t\ in} \\ \quad \mathbf{let\ x_c} = \llbracket \mathsf{\Delta,\Gamma\vdash e:\tau}\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\ \mathbf{in} \\ \quad\ \mathbf{x_t := x_c\ with\ x_k;} \\ \quad\ \mathbf{let\ x = \langle x_t, x_k\rangle\ in} \\ \qquad \llbracket \mathsf{C,\Delta,\Gamma;x:Ref\ \tau\vdash s}\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \end{array} & \text{otherwise} \end{cases}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\text{-New-nonat})$$

8.3 Properties of the L^{τ}-$\mathbf{L}^{\pi}$ Compiler

Theorem 6 (Compiler $\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is *CC*)**.** $\vdash \llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : CC$

Theorem 7 (Compiler $\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is *RSC*)**.** $\vdash \llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : RSC$

8.4 Cross-language Relation $\approxeq_{\beta}$

We define a more lenient relation on states $\approxeq_{\beta}$ analogous to $\approx_{\beta}$ (Rule Related states – Whole) but that ensures that all target locations that are related to secure source ones only vary accordingly: i.e., the attacker cannot change them.

$$\boxed{\mathsf{\Omega} \approxeq_{\beta} \mathbf{\Omega}}$$

$$\dfrac{\mathsf{H'} = \{\mathsf{\ell \mapsto v : \tau} \mid \mathsf{\ell \mapsto v : \tau \in H} \text{ and } \tau \nvdash \circ\}}{\vdash \mathtt{secure}(\mathsf{H}) = \mathsf{H'}}\ (\mathsf{L}^{\tau}\text{-Secure heap})$$

$$\dfrac{\nexists \ell \in \mathtt{secure}(\mathsf{H}) \qquad \ell \approx_{\beta} \langle \mathbf{n}, \_\rangle \qquad \mathbf{n} \in \mathtt{dom}(\mathbf{H})}{\mathsf{H}, \mathbf{H} \vdash \mathtt{low\text{-}loc}(\mathbf{n})}\ (\mathbf{L}^{\pi}\text{-Low Location})$$

$$\dfrac{\ell \in \mathtt{secure}(\mathsf{H}) \qquad \ell \approx_{\beta} \langle \mathbf{n}, \mathbf{k}\rangle \qquad \mathbf{n} \mapsto \_ : \mathbf{k} \in \mathbf{H}}{\mathsf{H}, \mathbf{H} \vdash \mathtt{high\text{-}loc}(\mathbf{n}) = \ell, \mathbf{k}}\ (\mathbf{L}^{\pi}\text{-High Location})$$

$$\dfrac{\ell \in \mathtt{secure}(\mathsf{H}). \ell \approx_{\beta} \langle \mathbf{n}, \mathbf{k}\rangle \qquad \mathbf{n} \mapsto \_ : \mathbf{k} \in \mathbf{H}}{\mathsf{H}, \mathbf{H} \vdash \mathtt{high\text{-}cap}(\mathbf{k})}\ (\mathbf{L}^{\pi}\text{-High Capability})$$

$$\text{(Related states – Secure)}$$
$$\frac{\begin{array}{c}\Omega = \Delta; \overline{\mathsf{F}}, \overline{\mathsf{F}'}; \bar{\mathsf{I}}; \mathsf{H} \triangleright \Pi \qquad \boldsymbol{\Omega} = \mathbf{H_0}; \overline{\mathbf{F}}, [\![\overline{\mathsf{F}'}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}; \bar{\mathbf{I}}; \mathbf{H} \triangleright \boldsymbol{\Pi} \qquad \Delta \vdash_\beta \mathbf{H_0} \\ \forall \mathbf{k}, \mathbf{n}, \ell.\ \text{if } \mathsf{H}, \mathbf{H} \vdash \texttt{high-loc}(\mathbf{n}) = \ell, \mathbf{k} \text{ then} \\ (1)\ \forall \pi \in \boldsymbol{\Pi} \text{ if } \mathsf{C} \vdash \pi : \mathbf{attacker} \text{ then } \mathbf{k} \notin \texttt{fv}(\pi) \\ (2)\ \forall \mathbf{n}' \mapsto \mathbf{v} : \eta \in \mathbf{H}, \\ (2a) \text{ if } \eta = \mathbf{k} \text{ then } \mathbf{n} = \mathbf{n}' \text{ and } \ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle \text{ and } \ell \mapsto \mathsf{v} : \tau \in \mathsf{H} \text{ and } \mathsf{v} \approx_\beta \mathbf{v} \\ (2b) \text{ if } \eta \neq \mathbf{k} \text{ then } \mathsf{H}, \mathbf{H} \vdash \texttt{low-loc}(\mathbf{n}') \text{ and } \forall \mathbf{k}'. \mathsf{H}, \mathbf{H} \vdash \texttt{high-cap}(\mathbf{k}'), \mathbf{v} \neq \mathbf{k}' \end{array}}{\Omega \approx_\beta \boldsymbol{\Omega}}$$

There is no $\texttt{secure}(\,\cdot\,)$ function for the target because they would be all locations that are related to a source location that itself is secure in the source. An alternative is to define $\texttt{secure}(\,\cdot\,)$ as all locations protected by a key $\mathbf{k}$ but the point of $\texttt{secure}(\,\cdot\,)$ is to setup the invariant to ensure the proof hold, so this alternative would be misleading.

Rule $\mathbf{L}^{\pi}$-Low Location tells when a target location is not secure. That is, when there is no secure source location that is related to it. This can be because the source location is not secure or because the relation does not exist, as in order for it to exist the triple must be added to β and we only add the triple for secure locations.

The intuition behind Rule Related states – Secure is that two states are related if the set of locations they monitor is related and then: for any target location $\mathbf{n}$ that is high (i.e., it has a related source counterpart ℓ whose type is secure and that is protected with a capability $\mathbf{k}$ that we call a high capability), then we have: (1) the capability $\mathbf{k}$ used to lock it is not in in any attacker code; (2) for any target level location $\mathbf{n}'$: (2a) either it is locked with a high capability $\mathbf{k}$ (i.e., a capability used to hide a high location) thus $\mathbf{n}'$ is also high, in which case it is related to a source location ℓ and the values v, $\mathbf{v}$ they point to are related; or (2b) it is not locked with a high capability, so we can derive that $\mathbf{n}'$ is a low location and its content $\mathbf{v}$ is not any high capability $\mathbf{k}'$.

♠

Lemma 5 (A target location is either high or low)**.**

$$\begin{array}{rl} & \forall \\ \text{if} & \mathsf{H} \approx_\beta \mathbf{H} \\ & \mathbf{n} \mapsto \mathbf{v} : \eta \in \mathbf{H} \\ \text{then either} & \mathsf{H}, \mathbf{H} \vdash \texttt{low-loc}(\mathbf{n}) \\ \text{or} & \exists \ell \in \texttt{dom}(\mathsf{H}). \\ & \mathsf{H}, \mathbf{H} \vdash \texttt{high-loc}(\mathbf{n}) = \ell, \eta \end{array}$$

Proof. Trivial, as Rule $\mathbf{L}^{\pi}$-Low Location and Rule $\mathbf{L}^{\pi}$-High Location are duals. □

9 *RSC*: Third Instance with Target Memory Isolation

Both compilers presented so far used a capability-based target language. To avoid giving the false impression that *RSC* is only useful for this kind of a target, we show here how to attain *RSC* when the protection mechanism in the target is completely different. We consider a new target language, L^I, which does not have capabilities, but instead offers coarse-grained memory isolation based on *enclaves*. This mechanism is supported (in hardware) in mainstream x86-64 and ARM CPUs (Intel calls this SGX [11]; ARM calls it TrustZone [17]). This is also straightforward to implement purely in software using any physical, VM-based, process-based, or in-process isolation technique. This section provides a high-level discussion on how to devise compiler $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ from our source language L^τ to L^I and why it attains *RSC*. Full formal details are presented in subsequent sections.

9.1 L^I, a Target Language with Memory Isolation

Language L^I replaces $\mathbf{L}^\pi$'s capabilities with a simple security abstraction called an enclave. An enclave is a collection of code and memory locations, with the properties that: (a) only code within the enclave can access the memory locations of the enclave, and (b) Code from outside can transfer control only to designated entry points in the enclave's code. For simplicity, L^I supports only one enclave. Generalizing this to many enclaves is straightforward.

To model the enclave, a L^I program has an additional component $\overline{E}$, the list of functions that reside in the enclave. A component thus has the form $C ::= H_0; \overline{F}; \overline{I}; \overline{E}$. Only functions that are listed in $\overline{E}$ can create ($let\ x = newiso\ e\ in\ s$), read ($!e$) and write ($x := e$) locations in the enclave. Locations in L^I are *integers* (not natural numbers). By convention, non-negative locations are outside the enclave (accessible from any function), while negative locations are inside the enclave (accessible only from functions in $\overline{E}$). The semantics are almost those of $\mathbf{L}^\pi$, but the expression semantics change to $\mathbf{C}; \mathbf{H}; \mathbf{f} \triangleright \mathbf{e} \hookrightarrow\!\!\!\!\rightarrow \mathbf{v}$, recording which function f is currently executing. The operational rule for any memory operation checks that either the access is to a location outside the enclave or that $f \in \overline{E}$ (formalized by $C \vdash f : prog$). Monitors of L^I are the same as those of $\mathbf{L}^\pi$.

9.2 Compiler from L^τ to L^I

The high-level structure of the compiler $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ is similar to that of $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$. $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ ensures that all the (and only the) functions of the (trusted) component we write are part of the enclave, i.e., constitute $\overline{E}$ (first rule below). Additionally, the compiler populates the safety-relevant heap H_0 based on the information in Δ (captured by the judgement $\Delta \vdash H_0$, whose details we elide here). Importantly, $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ also ensures that trusted locations are stored in the enclave. As before, the compiler relies on typing information for this. Locations whose types are shareable (subtypes of UN) are placed outside the enclave while those that trusted (not subtypes of UN) are placed inside.

As mentioned, $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ also attains RSC. The intuition is simple: all trusted locations (including safety-relevant locations) are in the enclave and adversarial code cannot tamper with them. The proof follows the proof of the previous compiler: We build a cross-language relation, which we show to be an invariant on executions of source and corresponding compiled programs. The only change is that every location in the *trusted target heap* is isolated in the enclave.

10 The Second Target Language: L^I

For clarity, we use a *pink, italics* font for L^I.

10.1 Syntax

$$
\begin{aligned}
\textit{Whole Programs}\ P &::= H_0; \overline{F}; \overline{I}; \overline{E}\\
\textit{Components}\ C &::= H_0; \overline{F}; \overline{I}; \overline{E}\\
\textit{Contexts}\ A &::= \overline{F}\,[\cdot]\\
\textit{Interfaces}\ I &::= f\\
\textit{Enclave functions}\ E &::= f\\
\textit{Functions}\ F &::= f(x) \mapsto s; return;\\
\textit{Operations}\ \oplus &::= + \mid -\\
\textit{Comparison}\ \otimes &::= == \mid < \mid >\\
\textit{Values}\ v &::= n \in \mathbb{Z} \mid \langle v, v\rangle \mid k\\
\textit{Expressions}\ e &::= x \mid v \mid e \oplus e \mid e \otimes e \mid \langle e, e\rangle \mid e.1 \mid e.2 \mid !e\\
\textit{Statements}\ s &::= skip \mid s; s \mid let\ x = e\ in\ s \mid ifz\ e\ then\ s\ else\ s \mid call\ f\ e\\
&\quad \mid\ \mid (\| s) \mid destruct\ x = e\ as\ B\ in\ s\ or\ s\\
&\quad \mid x := e \mid let\ x = new\ e\ in\ s \mid let\ x = newiso\ e\ in\ s\\
\textit{Patterns}\ B &::= nat \mid pair\\
\textit{Eval. Ctxs.}\ E &::= [\cdot] \mid e \oplus E \mid E \oplus n \mid e \otimes E \mid E \otimes n \mid !E\\
&\quad \mid \langle e, E\rangle \mid \langle E, v\rangle \mid E.1 \mid E.2\\
\textit{Heaps}\ H &::= \varnothing \mid H; n \mapsto v\\
\textit{Monitors}\ M &::= (\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c)\\
\textit{Mon. States}\ \sigma &\in \mathcal{S}\\
\textit{Mon. Reds.}\ \rightsquigarrow &::= \varnothing \mid \rightsquigarrow; (s, H, s)\\
\textit{Substitutions}\ \rho &::= \varnothing \mid \rho[v\ /\ x]\\
\textit{Single Process}\ \pi &::= (s)_{\overline{f}}\\
\textit{Processes}\ \Pi &::= \varnothing \mid \Pi \parallel \pi
\end{aligned}
$$

$$\begin{aligned}
\mathit{Prog.\ States}\ \Omega &::= C, H \triangleright \Pi \\
\mathit{Labels}\ \lambda &::= \epsilon \mid \alpha \\
\mathit{Actions}\ \alpha &::= \texttt{call}\ f\ \ v\ \ H? \mid \texttt{call}\ f\ \ v\ \ H! \mid \texttt{ret}\ H! \mid \texttt{ret}\ H? \\
\mathit{Traces}\ \overline{\alpha} &::= \varnothing \mid \overline{\alpha} \cdot \alpha
\end{aligned}$$

10.2 Operational Semantics of L^I

$\boxed{\text{Helpers}}$

$$\text{(}L^I\text{-Jump-Internal)}\quad \frac{\begin{array}{c}((f' \in \overline{I} \wedge f \in \overline{I}) \vee \\ (f' \notin \overline{I} \wedge f \notin \overline{I}))\end{array}}{\overline{I} \vdash f, f' : \mathit{internal}}
\qquad
\text{(}L^I\text{-Jump-IN)}\quad \frac{f \in \overline{I} \wedge f' \notin \overline{I}}{\overline{I} \vdash f, f' : \mathit{in}}
\qquad
\text{(}L^I\text{-Jump-OUT)}\quad \frac{f \notin \overline{I} \wedge f' \in \overline{I}}{\overline{I} \vdash f, f' : \mathit{out}}$$

$$\text{(}L^I\text{-prog-execs)}\quad \frac{C = H_0; \overline{F}; \overline{I}; \overline{E} \qquad f \in \overline{E}}{C \vdash f : \mathit{prog}}$$

$$\text{(}L^I\text{-Plug)}\quad \frac{\begin{array}{cc} A \equiv \overline{F}\,[\cdot] & C \equiv H_0; \overline{F'}; \overline{I}; \overline{E} \\ \vdash C, \overline{F} : \mathit{whole} & \mathit{main}(x) \mapsto s; \mathit{return}; \in \overline{F} \end{array}}{A\,[C] = H_0; \overline{F; F'}; \overline{I}; \overline{E}}$$

$$\text{(}L^I\text{-Whole)}\quad \frac{\begin{array}{c} C \equiv H_0; \overline{F'}; \overline{I}; \overline{E} \\ \texttt{names}(\overline{F}) \cap \texttt{names}(\overline{F'}) = \varnothing \qquad \texttt{names}(\overline{I}) \subseteq \texttt{names}(\overline{F}) \qquad \forall n \mapsto v \in H_0, n < 0 \end{array}}{\vdash C, \overline{F} : \mathit{whole}}$$

$$\text{(}L^I\text{-Initial State)}\quad \frac{P \equiv H_0; \overline{F}; \overline{I}; \overline{E} \qquad \mathit{main}(x) \mapsto s; \mathit{return}; \in \overline{F}}{\Omega_0\,(P) = P; H_0 \triangleright (s[0\ /\ x])_{\mathit{main}}}$$

10.2.1 Component Semantics

$C; H; \overline{f} \triangleright e \ \hookrightarrow\!\!\!\!\rightarrow\ e'$	Expression e reduces to e'.
$C, H \triangleright \Pi \xrightarrow{\epsilon} C', H' \triangleright \Pi'$	Processes Π reduce to Π' and evolve the rest accordingly. emitting label λ.
$\Omega \xRightarrow{\overline{\alpha}} \Omega'$	Program state Ω steps to Ω' emitting trace $\overline{\alpha}$.

$\boxed{C; H; \overline{f} \triangleright e \ \hookrightarrow\!\!\!\!\rightarrow\ e'}$

(EL^{I}-val)

$$\frac{}{C; H; \overline{f} \triangleright v \hookrightarrow\!\!\!\twoheadrightarrow v}$$

(EL^{I}-p1)

$$\frac{}{C; H; \overline{f} \triangleright \langle v, v' \rangle .\mathit{1} \hookrightarrow\!\!\!\twoheadrightarrow v}$$

(EL^{I}-p2)

$$\frac{}{C; H; \overline{f} \triangleright \langle v, v' \rangle .\mathit{1} \hookrightarrow\!\!\!\twoheadrightarrow v'}$$

(EL^{I}-op)

$$\frac{n \oplus n' = n''}{C; H; \overline{f} \triangleright n \oplus n' \hookrightarrow\!\!\!\twoheadrightarrow n''}$$

(EL^{I}-comp)

$$\frac{\text{if } n \otimes n' = \text{true then } n'' = \mathit{0} \text{ else } n'' = \mathit{1}}{C; H; \overline{f} \triangleright n \otimes n' \hookrightarrow\!\!\!\twoheadrightarrow n''}$$

(EL^{I}-deref)

$$\frac{n \mapsto v \in H \qquad n \geq \mathit{0}}{C; H; \overline{f} \triangleright !n \hookrightarrow\!\!\!\twoheadrightarrow v}$$

(EL^{I}-deref-iso)

$$\frac{n \mapsto v \in H \qquad n < \mathit{0} \qquad C \vdash f : \mathit{prog}}{C; H; \overline{f; f} \triangleright !n \hookrightarrow\!\!\!\twoheadrightarrow v}$$

(EL^{I}-ctx)

$$\frac{C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow e'}{C; H; \overline{f} \triangleright E\,[e] \hookrightarrow\!\!\!\twoheadrightarrow E\,[e']}$$

$$\boxed{C; H \triangleright s \xrightarrow{\lambda} C'; H' \triangleright s'}$$

We elide the suffix with the stack of functions when obvious.

(EL^{I}-sequence)

$$\frac{}{C, H \triangleright \mathit{skip}; s \xrightarrow{\epsilon} C, H \triangleright s}$$

(EL^{I}-step)

$$\frac{C, H \triangleright s \xrightarrow{\lambda} C, H \triangleright s'}{C, H \triangleright s; s'' \xrightarrow{\lambda} C, H \triangleright s'; s}$$

(EL^{I}-if-true)

$$\frac{C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow \mathit{0}}{C, H \triangleright \mathit{ifz}\ e\ \mathit{then}\ s\ \mathit{else}\ s' \xrightarrow{\epsilon} C, H \triangleright s}$$

(EL^{I}-if-false)

$$\frac{C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow n \qquad n \not\equiv \mathit{0}}{C, H \triangleright \mathit{ifz}\ e\ \mathit{then}\ s\ \mathit{else}\ s' \xrightarrow{\epsilon} C, H \triangleright s'}$$

(EL^{I}-letin)

$$\frac{C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow v}{C, H \triangleright \mathit{let}\ x = e\ \mathit{in}\ s \xrightarrow{\epsilon} C, H \triangleright s[v \,/\, x]}$$

(EL^{I}-new)

$$\frac{H = H_{\mathit{1}}; n \mapsto \_ \qquad C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow v}{C, H \triangleright \mathit{let}\ x = \mathit{new}\ e\ \mathit{in}\ s \xrightarrow{\epsilon} C, H; n + \mathit{1} \mapsto v \triangleright s[n + \mathit{1} \,/\, x]}$$

(EL^{I}-isolate)

$$\frac{\begin{array}{cc} H = n \mapsto \_; H_{\mathit{1}} & C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow v \\ \overline{f} = \overline{f'} \cdot f & C \vdash f : \mathit{prog} \end{array}}{C, H \triangleright (\mathit{let}\ x = \mathit{newiso}\ e\ \mathit{in}\ s)_{\overline{f}} \xrightarrow{\epsilon} C, n - \mathit{1} \mapsto v; H \triangleright (s[n - \mathit{1} \,/\, x])_{\overline{f}}}$$

(EL^{I}-assign)

$$\frac{\begin{array}{c} C; H; \overline{f} \triangleright e \hookrightarrow\!\!\!\twoheadrightarrow v \\ H = H_{\mathit{1}}; n \mapsto \_; H_{\mathit{2}} \qquad H' = H_{\mathit{1}}; n \mapsto v; H_{\mathit{2}} \qquad n \geq \mathit{0} \end{array}}{C, H \triangleright n := e \xrightarrow{\epsilon} C, H' \triangleright \mathit{skip}}$$

(EL^I-assign-iso)

$$\frac{\begin{array}{ccc} C;H;\overline{f}\triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow v & \overline{f}=\overline{f'}\cdot f & C\vdash f : \mathit{prog} \\ H = H_1;n\mapsto \_;H_2 & H'=H_1;n\mapsto v;H_2 & n<0\end{array}}{C,H\triangleright(n := e)_{\overline{f}} \xrightarrow{\epsilon} C,H'\triangleright(\mathit{skip})_{\overline{f}}}$$

(EL^I-call-internal)

$$\frac{\begin{array}{cc} \overline{C}.\mathtt{intfs}\vdash f,f' : \mathit{internal} & \overline{f'}=\overline{f''};f' \\ f(x)\mapsto s;\mathit{return};\in C.\mathtt{funs} & C;H;\overline{f}\triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow v\end{array}}{C,H\triangleright(\mathit{call}\ f\ \ e)_{\overline{f'}} \xrightarrow{\epsilon} C,H\triangleright(s;\mathit{return};[v\ /\ x])_{\overline{f'};f}}$$

(EL^I-callback)

$$\frac{\begin{array}{cc} \overline{f'}=\overline{f''};f' & f(x)\mapsto s;\mathit{return};\in\overline{F} \\ \overline{C}.\mathtt{intfs}\vdash f',f : \mathit{out} & C;H;\overline{f}\triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow v\end{array}}{C,H\triangleright(\mathit{call}\ f\ \ e)_{\overline{f'}} \xrightarrow{\mathtt{call}\ f\ \ v\ \ H!} C,H\triangleright(s;\mathit{return};[v\ /\ x])_{\overline{f'};f}}$$

(EL^I-call)

$$\frac{\begin{array}{cc} \overline{f'}=\overline{f''};f' & f(x)\mapsto s;\mathit{return};\in C.\mathtt{funs} \\ \overline{C}.\mathtt{intfs}\vdash f',f : \mathit{in} & C;H;\overline{f}\triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow v\end{array}}{C,H\triangleright(\mathit{call}\ f\ \ e)_{\overline{f'}} \xrightarrow{\mathtt{call}\ f\ \ v\ \ H?} C,H\triangleright(s;\mathit{return};[v\ /\ x])_{\overline{f'};f}}$$

(EL^I-reo-internal)

$$\frac{\overline{C}.\mathtt{intfs}\vdash f,f' : \mathit{internal} \qquad \overline{f'}=\overline{f''};f'}{C,H\triangleright(\mathit{return};)_{\overline{f'};f} \xrightarrow{\epsilon} C,H\triangleright(\mathit{skip})_{\overline{f'}}}$$

(EL^I-retback)

$$\frac{\overline{C}.\mathtt{intfs}\vdash f,f' : \mathit{in} \qquad \overline{f'}=\overline{f''};f'}{C,H\triangleright(\mathit{return};)_{\overline{f'};f} \xrightarrow{\mathtt{ret}\ \ H?} C,H\triangleright(\mathit{skip})_{\overline{f'}}}$$

(EL^I-return)

$$\frac{\overline{C}.\mathtt{intfs}\vdash f,f' : \mathit{out} \qquad \overline{f'}=\overline{f''};f'}{C,H\triangleright(\mathit{return};)_{\overline{f'};f} \xrightarrow{\mathtt{ret}\ \ H!} C,H\triangleright(\mathit{skip})_{\overline{f'}}}$$

(EL^I-destruct-nat)

$$\frac{C;H;\overline{f}\triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow n}{C,H\triangleright \mathit{destruct}\ x = e\ \mathit{as}\ \mathit{nat}\ \mathit{in}\ s\ \mathit{or}\ s' \xrightarrow{\epsilon} C,H\triangleright s[n\ /\ x]}$$

(EL^I-destruct-pair)

$$\frac{C;H;\overline{f}\triangleright e \hookrightarrow\!\!\!\!\twoheadrightarrow \langle v,v'\rangle}{C,H\triangleright \mathit{destruct}\ x = e\ \mathit{as}\ \mathit{pair}\ \mathit{in}\ s\ \mathit{or}\ s' \xrightarrow{\epsilon} C,H\triangleright s[\langle v,v'\rangle\ /\ x]}$$

(EL^I-destruct-not)

$$\frac{\text{otherwise}}{C,H\triangleright \mathit{destruct}\ x = e\ \mathit{as}\ B\ \mathit{in}\ s\ \mathit{or}\ s' \xrightarrow{\epsilon} C,H\triangleright s'}$$

$$\boxed{C,H\triangleright \Pi \hookrightarrow\!\!\!\!\twoheadrightarrow C',H'\triangleright \Pi'}$$

$$\textsf{(E}L^{I}\textsf{-par)}\quad \dfrac{\begin{array}{c}\Pi = \Pi_1 \parallel (s)_{\overline{f}} \parallel \Pi_2 \\ \Pi' = \Pi_1 \parallel (s')_{\overline{f'}} \parallel \Pi_2 \\ C, H \triangleright (s)_{\overline{f}} \hookrightarrow\!\!\!\!\rightarrow C', H' \triangleright (s')_{\overline{f'}}\end{array}}{C, H \triangleright \Pi \hookrightarrow\!\!\!\!\rightarrow C', H' \triangleright \Pi'}
\qquad
\textsf{(E}L^{I}\textsf{-fork)}\quad \dfrac{\begin{array}{c}\Pi = \Pi_1 \parallel ((\parallel s))_{\overline{f;f}} \parallel \Pi_2 \\ \Pi' = \Pi_1 \parallel (0)_{\overline{f;f}} \parallel \Pi_2 \parallel (s)_{f}\end{array}}{C, H \triangleright \Pi \hookrightarrow\!\!\!\!\rightarrow C, H \triangleright \Pi'}$$

$$\boxed{\Omega \xRightarrow{\overline{\alpha}} \Omega'}$$

$$\textsf{(E}L^{I}\textsf{-single)}\quad \dfrac{\Omega \xrightarrow{\alpha} \Omega'}{\Omega \xRightarrow{\alpha} \Omega'}
\qquad
\textsf{(E}L^{I}\textsf{-silent)}\quad \dfrac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Longrightarrow \Omega'}
\qquad
\textsf{(E}L^{I}\textsf{-trans)}\quad \dfrac{\begin{array}{c}\Omega \xRightarrow{\overline{\alpha}} \Omega'' \\ \Omega'' \xRightarrow{\overline{\alpha'}} \Omega'\end{array}}{\Omega \xRightarrow{\overline{\alpha}\cdot\overline{\alpha'}} \Omega'}$$

10.3 Monitor Semantics

$$\boxed{\texttt{mon-care}(\,\cdot\,)}$$

$$\textsf{(}L^{I}\textsf{-Monitor-related heap)}\quad \dfrac{H' = \{n \mapsto v : \eta \mid n \in \texttt{dom}(H_0) \text{ and } n \mapsto v : \eta \in H\}}{\texttt{mon-care}(H, H_0) = H'}$$

$$\boxed{M; H \rightsquigarrow M'}$$

$$\textsf{(}L^{I}\textsf{-Monitor Step)}\quad \dfrac{\begin{array}{c}M = (\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c) \qquad M' = (\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_f) \\ (s_c, \texttt{mon-care}(H, H_0), s_f) \in \rightsquigarrow\end{array}}{M; H \rightsquigarrow M'}$$

$$\textsf{(}L^{I}\textsf{-Monitor Step Trace Base)}\quad \dfrac{}{M; \varnothing \rightsquigarrow M}
\qquad
\textsf{(}L^{I}\textsf{-Monitor Step Trace)}\quad \dfrac{M; \overline{H} \rightsquigarrow M'' \qquad M''; H \rightsquigarrow M'}{M; \overline{H} \cdot H \rightsquigarrow M'}$$

$$\textsf{(}L^{I}\textsf{-valid trace)}\quad \dfrac{M; \overline{H} \rightsquigarrow M'}{M \vdash \overline{\texttt{mon}\ H}}$$

10.4 Monitor Agreement for L^{I}

Definition 22 (L^{I}: $M \frown C$)**.**

$$(\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c) \frown (H_0; \overline{F}; \overline{I}; \overline{E})$$

A monitor and a component agree if they focus on the same set of locations H_0.

10.5 Properties of L^I

Definition 23 (L^I Attacker)**.**

$$C \vdash_{att} A \overset{\mathsf{def}}{=} C = H_0; \overline{F}; \overline{I}; \overline{E}, A = \overline{F'}, \texttt{names}(\overline{F}) \cap \texttt{names}(\overline{F'}) = \varnothing$$

$$C \vdash_{att} \pi \overset{\mathsf{def}}{=} \pi = (s)_{\overline{f};f} \text{ and } f \in C.\texttt{itfs}$$

$$C \vdash_{att} \Pi \longrightarrow \Pi' \overset{\mathsf{def}}{=} \Pi = \Pi_1 \parallel \pi \parallel \Pi_2 \text{ and } \Pi' = \Pi_1 \parallel \pi' \parallel \Pi_2$$
$$\text{and } C \vdash_{att} \pi \text{ and } C \vdash_{att} \pi'$$

11 Second Compiler from L^τ to L^I

For this compiler we need a different partial bijection, which we indicate with φ and its type is $\ell \times n$. It has the same properties of β listed in Section 3.3.

The cross-language relation $\approx$ is unchanged but for the relation of locations, as they are no longer compiled as pairs:

- $\ell \approx_\varphi n$ if $(\ell, n) \in \varphi$

Actions relation is unchanged from Rule Call relation etc.

Heaps relation is unchanged (modulo the elision of capabilities) from Rule Heap relation.

Process relation is unchanged from Rule Single process relation etc.

State relation is unchanged from Rule Related states – Whole.

The monitor relation $\mathsf{M} \approx M$ is defined as in Rule Monitor relation .

Some auxiliary functions are changed:

$$\boxed{\Delta \vdash_\varphi H_0 \quad \Delta, H \vdash v : \tau}$$

$$\text{(Initial-heap)}\quad \frac{\Delta \vdash H \qquad \Delta, H \vdash v : \tau \qquad \ell \approx_\varphi n}{\Delta, \ell : \tau \vdash_\varphi H; n \mapsto v}$$

$$\text{(Initial-value)}\quad \frac{\begin{array}{c}(\tau \equiv \mathsf{Bool} \wedge v \equiv 0) \qquad \vee \qquad (\tau \equiv \mathsf{Nat} \wedge v \equiv 0) \qquad \vee \\ (\tau \equiv \mathsf{Ref}\ \tau \wedge v \equiv n' \wedge n' \mapsto v' \in H \wedge \ell' \approx_\varphi n' \wedge \ell : \tau \in \Delta, \Delta, H \vdash v' : \tau) \qquad \vee \\ (\tau \equiv \tau_1 \times \tau_2 \wedge v \equiv \langle v_1, v_2 \rangle \wedge \Delta, H \vdash v_1 : \tau_1 \wedge \Delta, H \vdash v_2 : \tau_2)\end{array}}{\Delta, H \vdash_\varphi v : \tau}$$

Definition 24 (Compiler L^τ to L^I)**.** $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I} : \mathsf{C} \to C$

Given that $\mathsf{C} = \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}$ if $\vdash \mathsf{C} : \mathsf{UN}$ then $\llbracket \mathsf{C} \rrbracket^{\mathsf{L}^\tau}_{L^I}$ is defined as follows:

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-component})\\ \mathsf{C} \equiv \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}} \\ \mathsf{C} \vdash \overline{\mathsf{F}} : \mathsf{UN} \\ \mathtt{names}(\overline{\mathsf{F}}) \cap \mathtt{names}(\overline{\mathsf{I}}) = \varnothing \\ \Delta \vdash \mathsf{ok}\end{array}}{\vdash \mathsf{C} : \mathsf{UN}} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = H_0; \llbracket \overline{\mathsf{F}} \rrbracket^{\mathsf{L}^\tau}_{L^I}; \llbracket \overline{\mathsf{I}} \rrbracket^{\mathsf{L}^\tau}_{L^I}; \mathtt{dom}(\overline{\mathsf{F}}) \qquad \text{if } \Delta \vdash_{\varphi_0} H_0$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Component$)$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-function})\\ \mathsf{F} \equiv \mathsf{f}(\mathsf{x} : \mathsf{UN}) \mapsto \mathsf{s}; \mathsf{return}; \\ \mathsf{C}, \Delta; \mathsf{x} : \mathsf{UN} \vdash \mathsf{s} \\ \forall \mathsf{f} \in \mathtt{fn}(s), \mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{funs}) \\ \forall \mathsf{f} \in \mathtt{dom}(\mathsf{C}.\mathtt{intfs})\end{array}}{\mathsf{C} \vdash \mathsf{F} : \mathsf{UN}} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = f(x) \mapsto \llbracket \mathsf{C}; \Delta; \mathsf{x} : \mathsf{UN} \vdash \mathsf{s} \rrbracket^{\mathsf{L}^\tau}_{L^I}; return;$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Function$)$

$$\llbracket \mathsf{f} \rrbracket^{\mathsf{L}^\tau}_{L^I} = f$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Interfaces$)$

Expressions

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-true})\\ \Delta, \Gamma \vdash \diamond\end{array}}{\Delta, \Gamma \vdash \mathsf{true} : \mathsf{Bool}} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = 0$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-True$)$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-false})\\ \Delta, \Gamma \vdash \diamond\end{array}}{\Delta, \Gamma \vdash \mathsf{false} : \mathsf{Bool}} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = 1$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-False$)$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-nat})\\ \Delta, \Gamma \vdash \diamond\end{array}}{\Delta, \Gamma \vdash \mathsf{n} : \mathsf{Nat}} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = n$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Nat$)$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-var})\\ \mathsf{x} : \tau \in \Gamma\end{array}}{\Delta, \Gamma \vdash \mathsf{x} : \tau} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = x$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Var$)$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-loc})\\ \ell : \tau \in \Delta\end{array}}{\Delta, \Gamma \vdash \ell : \tau} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = n$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Loc$)$

$$\left\llbracket \dfrac{\begin{array}{c}(\mathsf{TL}^\tau\text{-pair})\\ \Delta, \Gamma \vdash \mathsf{e}_1 : \tau \\ \Delta, \Gamma \vdash \mathsf{e}_2 : \tau'\end{array}}{\Delta, \Gamma \vdash \langle \mathsf{e}_1, \mathsf{e}_2 \rangle : \tau \times \tau'} \right\rrbracket^{\mathsf{L}^\tau}_{L^I} = \left\langle \llbracket \Delta, \Gamma \vdash \mathsf{e}_1 : \tau \rrbracket^{\mathsf{L}^\tau}_{L^I}, \llbracket \Delta, \Gamma \vdash \mathsf{e}_2 : \tau' \rrbracket^{\mathsf{L}^\tau}_{L^I} \right\rangle$$

$(\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{L^I}$-Pair$)$

$$\left\llbracket \frac{\overset{(\mathsf{TL}^{\tau}\text{-proj-1})}{\Delta,\Gamma\vdash \mathsf{e}:\tau\times\tau'}}{\Delta,\Gamma\vdash \mathsf{e.1}:\tau} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\tau\times\tau'\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}.\mathit{1} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-P1})$$

$$\left\llbracket \frac{\overset{(\mathsf{TL}^{\tau}\text{-proj-2})}{\Delta,\Gamma\vdash \mathsf{e}:\tau\times\tau'}}{\Delta,\Gamma\vdash \mathsf{e.2}:\tau'} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\tau\times\tau'\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}.\mathit{2} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-P2})$$

$$\left\llbracket \frac{\overset{(\mathsf{TL}^{\tau}\text{-dereference})}{\Delta,\Gamma\vdash \mathsf{e}:\mathsf{Ref}\ \tau}}{\Delta,\Gamma\vdash \mathsf{!e}:\tau} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = !\llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{Ref}\ \tau\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}.\mathit{1} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-Deref})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-op})\\ \Delta,\Gamma\vdash \mathsf{e}:\mathsf{Nat}\\ \Delta,\Gamma\vdash \mathsf{e}':\mathsf{Nat}\end{array}}{\Delta,\Gamma\vdash \mathsf{e}\oplus\mathsf{e}':\mathsf{Nat}} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{Nat}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} \oplus \llbracket \Delta,\Gamma\vdash \mathsf{e}':\mathsf{Nat}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-op})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-cmp})\\ \Delta,\Gamma\vdash \mathsf{e}:\mathsf{Nat}\\ \Delta,\Gamma\vdash \mathsf{e}':\mathsf{Nat}\end{array}}{\Delta,\Gamma\vdash \mathsf{e}\otimes\mathsf{e}':\mathsf{Bool}} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{Nat}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} \otimes \llbracket \Delta,\Gamma\vdash \mathsf{e}':\mathsf{Nat}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-cmp})$$

$$\left\llbracket \frac{\overset{(\mathsf{TL}^{\tau}\text{-coercion})}{\Delta,\Gamma\vdash \mathsf{e}:\tau \qquad \tau\vdash\circ}}{\Delta,\Gamma\vdash \mathsf{e}:\mathsf{UN}} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\tau\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-Coerce})$$

Statements

$$\left\llbracket \frac{(\mathsf{TL}^{\tau}\text{-skip})}{\mathsf{C},\Delta,\Gamma\vdash \mathsf{skip}} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \mathit{skip} \qquad (\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-Skip})$$

$$\left\llbracket \frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-new})\\ \mathsf{C},\Delta,\Gamma\vdash \mathsf{e}:\tau\\ \mathsf{C},\Delta,\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash \mathsf{let\ x = new}_{\tau}\ \mathsf{e\ in\ s}} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \begin{cases} \begin{array}{l} \mathit{let}\ x\ = \mathit{new}\ \llbracket \Delta,\Gamma\vdash\mathsf{e}:\tau\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \quad \mathit{in}\ \llbracket \mathsf{C},\Delta,\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} \end{array} & \text{if } \tau = \mathsf{UN}\\[1em] \begin{array}{l} \mathit{let}\ x\ = \mathit{newiso}\ \llbracket \Delta,\Gamma\vdash\mathsf{e}:\tau\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \quad \mathit{in}\ \llbracket \mathsf{C},\Delta,\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} \end{array} & \text{else} \end{cases}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-New})$$

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-function-call})\\ ((\mathsf{f}\in\mathtt{dom}(\mathsf{C}.\mathtt{funs}))\\ \vee(\mathsf{f}\in\mathtt{dom}(\mathsf{C}.\mathtt{intfs})))\\ \Delta,\Gamma\vdash\mathsf{e}:\mathsf{UN}\end{array}}{\Delta,\Gamma\vdash\mathsf{call\ f\ e}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=\mathit{call}\ f\ [\![\Delta,\Gamma\vdash\mathsf{e}:\mathsf{UN}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-call)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-if})\\ \Delta,\Gamma\vdash\mathsf{e}:\mathtt{Bool}\\ \mathsf{C},\Delta,\Gamma\vdash\mathsf{s_t}\\ \mathsf{C},\Delta,\Gamma\vdash\mathsf{s_e}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash\mathsf{if\ e\ then\ s_t\ else\ s_e}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=\begin{array}{l}\mathit{ifz}\ [\![\Delta,\Gamma\vdash\mathsf{e}:\mathtt{Bool}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\\ \quad\mathit{then}\ [\![\mathsf{C},\Delta,\Gamma\vdash\mathsf{s_t}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\\ \quad\mathit{else}\ [\![\mathsf{C},\Delta,\Gamma\vdash\mathsf{s_e}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\end{array}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-If)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-sequence})\\ \mathsf{C},\Delta,\Gamma\vdash\mathsf{s_u}\\ \mathsf{C},\Delta,\Gamma\vdash\mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash\mathsf{s_u};\mathsf{s}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=[\![\mathsf{C},\Delta,\Gamma\vdash\mathsf{s_u}]\!]^{\mathsf{L}^{\tau}}_{L^{I}};[\![\mathsf{C},\Delta,\Gamma;\Gamma'\vdash\mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-Seq)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-letin})\\ \Delta,\Gamma\vdash\mathsf{e}:\tau\\ \mathsf{C},\Delta,\Gamma;\mathsf{x}:\tau\vdash\mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash\mathsf{let\ x}:\tau=\mathsf{e\ in\ s}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=\begin{array}{l}\mathit{let}\ x{=}[\![\Delta,\Gamma\vdash\mathsf{e}:\tau]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\\ \quad\mathit{in}\ [\![\mathsf{C},\Delta,\Gamma;\mathsf{x}:\tau\vdash\mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\end{array}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-Letin)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-assign})\\ \Delta,\Gamma\vdash\mathsf{x}:\mathsf{Ref}\ \tau\\ \Delta,\Gamma\vdash\mathsf{e}:\tau\end{array}}{\mathsf{C},\Delta,\Gamma\vdash\mathsf{x}:=\mathsf{e}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=[\![\Delta,\Gamma\vdash\mathsf{x}:\mathsf{Ref}\ \tau]\!]^{\mathsf{L}^{\tau}}_{L^{I}}:=[\![\Delta,\Gamma\vdash\mathsf{e}:\tau]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-Assign)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-fork})\\ \mathsf{C},\Delta,\Gamma\vdash\mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash(\parallel\mathsf{s})}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=\left(\parallel[\![\mathsf{C},\Delta,\Gamma\vdash\mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\right)$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-Fork)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-process})\\ \mathsf{C},\Delta,\Gamma\vdash\mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash(\mathsf{s})_{\overline{\mathsf{f}}}}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=\left([\![\mathsf{C},\Delta,\Gamma\vdash\mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\right)_{[\![\overline{\mathsf{f}}]\!]^{\mathsf{L}^{\tau}}_{L^{I}}}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-Proc)

$$\left[\!\!\left[\frac{\begin{array}{c}(\mathsf{TL}^{\tau}\text{-soup})\\ \mathsf{C},\Delta,\Gamma\vdash\pi\\ \mathsf{C},\Delta,\Gamma\vdash\Pi\end{array}}{\mathsf{C},\Delta,\Gamma\vdash\pi\parallel\Pi}\right]\!\!\right]^{\mathsf{L}^{\tau}}_{L^{I}}=[\![\mathsf{C},\Delta,\Gamma\vdash\pi]\!]^{\mathsf{L}^{\tau}}_{L^{I}}\parallel[\![\mathsf{C},\Delta,\Gamma\vdash\Pi]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$$

$([\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$-Soup)

$$\left\llbracket \dfrac{\begin{array}{c}\text{(TL}^{\tau}\text{-endorse)}\\ \Delta,\Gamma\vdash \mathsf{e}:\mathsf{UN}\\ \mathsf{C},\Delta,\Gamma;(\mathsf{x}:\varphi)\vdash \mathsf{s}\end{array}}{\mathsf{C},\Delta,\Gamma\vdash \mathsf{endorse\ x}=\mathsf{e}\ \mathsf{as}\ \varphi\ \mathsf{in\ s}} \right\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} = \begin{cases} \begin{array}{l} \mathit{destruct}\ x \ = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{UN}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\ \mathit{as\ nat\ in}\\ \ \mathit{ifz}\ x\ \mathit{then}\\ \ \ \llbracket \mathsf{C},\Delta,\Gamma;(\mathsf{x}:\varphi)\vdash \mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \ \ \mathit{else\ ifz}\ x-\mathit{1}\ \mathit{then}\\ \ \ \ \llbracket \mathsf{C},\Delta,\Gamma;(\mathsf{x}:\varphi)\vdash \mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \ \ \ \mathit{else\ wrong}\\ \ \mathit{or\ wrong}\\ \text{if } \varphi=\mathsf{Bool}\\ \\ \mathit{destruct}\ x \ = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{UN}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\ \mathit{as\ nat\ in}\\ \llbracket \mathsf{C},\Delta,\Gamma;(\mathsf{x}:\varphi)\vdash \mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \mathit{or\ wrong}\\ \text{if } \varphi=\mathsf{Nat}\\ \\ \mathit{destruct}\ x \ = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{UN}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\ \mathit{as\ pair\ in}\\ \llbracket \mathsf{C},\Delta,\Gamma;(\mathsf{x}:\varphi)\vdash \mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \mathit{or\ wrong}\\ \text{if } \varphi=\mathsf{UN}\times\mathsf{UN}\\ \\ \mathit{destruct}\ x \ = \llbracket \Delta,\Gamma\vdash \mathsf{e}:\mathsf{UN}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\ \mathit{as\ nat\ in}\\ \ !x;\\ \ \llbracket \mathsf{C},\Delta,\Gamma;(\mathsf{x}:\varphi)\vdash \mathsf{s}\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\\ \mathit{or\ wrong}\\ \text{if } \varphi=\mathsf{Ref\ UN} \end{array} \end{cases}$$

$$(\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}\text{-Endorse})$$

We use *wrong* as before for **wrong**.

11.1 Properties of the L^{τ}-L^{I} Compiler

Theorem 8 (Compiler $\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}$ is *CC*)**.** $\vdash \llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} : CC$

Theorem 9 (Compiler $\llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}}$ is *RSC*)**.** $\vdash \llbracket\cdot\rrbracket^{\mathsf{L}^{\tau}}_{L^{I}} : RSC$

11.2 Cross-language Relation $\approx_{\varphi}$

As before, we define a more lenient relation on states $\approx_{\varphi}$

$$\boxed{\Omega \approx_{\varphi} \mathit{\Omega}}$$

$$\dfrac{\begin{array}{c}(L^{I}\text{-Low Location})\\ \nexists \ell \in \mathtt{secure}(\mathsf{H}) \qquad \ell \approx_{\varphi} n \qquad n \geq \mathit{0}\end{array}}{\mathsf{H}, H \vdash \mathtt{low\text{-}loc}(n)}$$

$$\dfrac{\begin{array}{c}(L^{I}\text{-High Location})\\ \ell \in \mathtt{secure}(\mathsf{H}) \qquad \ell \approx_{\varphi} n \qquad n < \mathit{0}\end{array}}{\mathsf{H}, H \vdash \mathtt{high\text{-}loc}(n) = \ell}$$

(Related states – Secure)

$$\frac{\begin{array}{c}\mathsf{\Omega} = \mathsf{\Delta}; \overline{\mathsf{F}}, \overline{\mathsf{F'}}; \overline{\mathsf{I}}; \mathsf{H} \triangleright \mathsf{\Pi} \qquad \mathit{\Omega} = H_0; \overline{F}, \llbracket \overline{\mathsf{F'}} \rrbracket^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}; \overline{I}; \overline{E}; H \triangleright \mathit{\Pi} \qquad \mathsf{\Delta} \vdash_{\varphi} H_0 \\ \forall n, \ell.\ \text{if } \mathsf{H}, H \vdash \texttt{high-loc}(n) = \ell \text{ then} \\ n \mapsto v \in H \text{ and } \ell \mapsto \mathsf{v} : \tau \in \mathsf{H} \text{ and } \mathsf{v} \approx_{\varphi} v\end{array}}{\mathsf{\Omega} \approxeq_{\varphi} \mathit{\Omega}}$$

We change the definition of a "high location" to be one that is in the enclave, i.e., whose address is less than 0.

The intuition behind Rule Related states – Secure is that high locations only need to be in sync, nothing is enforced on low locations. Compared to Rule Related states – Secure, we have less conditions because we don't have to track fine-grained capabilities but just if an address is part of the enclave or not.

♠

Lemma 6 (A L^I target location is either high or low)**.**

$$\begin{aligned} &\forall \\ \text{if } \quad &\mathsf{H} \approx_{\varphi} H \\ &n \mapsto v \in H \\ \text{then either } \quad &\mathsf{H}, H \vdash \texttt{low-loc}(n) \\ \text{or } \quad &\exists \ell \in \texttt{dom}(\mathsf{H}). \\ &\mathsf{H}, H \vdash \texttt{high-loc}(n) = \ell \end{aligned}$$

Proof. Trivial, as Rule L^I-Low Location and Rule L^I-High Location are duals. □

12 Proofs

12.1 Proof of Theorem 1 (*PF-RSC* and *RSC* are equivalent)

Proof. $\Rightarrow$ **HP**

$$\begin{aligned} \text{if } & \forall \mathbf{A}, \overline{\alpha}.\ [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} \vdash \mathbf{A} : \mathbf{attacker} \\ & \vdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right] : \mathbf{whole}\ \boldsymbol{\Omega}_{\mathbf{0}}\left([\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right) \xRightarrow{\overline{\alpha}} \boldsymbol{\Omega} \\ \text{then } & \exists \mathsf{A}, \overline{\alpha}\mathsf{C} \vdash \mathsf{A} : \mathsf{attacker} \\ & \vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}\ \Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \Omega \\ & \mathtt{heaps}(\overline{\alpha}) \approx_\beta \mathtt{heaps}(\overline{\alpha}) \end{aligned}$$

TH

$$\begin{aligned} \text{if } & \mathsf{M} \approx_\beta \mathbf{M} \\ & \forall \mathsf{A}, \overline{\alpha}.\ \vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole} \quad \text{if } \Omega_0\left(\mathsf{C}\right) \xRightarrow{\overline{\alpha}} \Omega \\ & \qquad\qquad\qquad\qquad \text{then } \mathsf{M} \vdash \overline{\alpha} \\ \text{then } & \forall \mathbf{A}, \overline{\alpha}.\ \vdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}}\right] : \mathbf{whole} \quad \text{if } \boldsymbol{\Omega}_{\mathbf{0}}\left([\![\mathsf{C}, \mathbf{M}]\!]^{\mathsf{S}}_{\mathbf{T}}\right) \xRightarrow{\overline{\alpha}} \boldsymbol{\Omega} \\ & \qquad\qquad\qquad\qquad \text{then } \mathbf{M} \vdash \overline{\alpha} \end{aligned}$$

We proceed by contradiction and assume that $\mathbf{M} \nvdash \overline{\alpha}$ while $\mathsf{C} \vdash \overline{\alpha}$.

By the relatedness of the traces, by Rules Call relation to Returnback relation we have $\mathsf{H} \approx_\beta \mathbf{H}$ for all heaps in the traces.

But if the heaps are related and the source steps (by unfolding $\mathsf{M} \vdash \overline{\alpha}$), then by point 3.b in Definition 8 ($\mathsf{M}\mathcal{R}\mathbf{M}$) we have that the target monitor also steps, so $\mathbf{M} \vdash \overline{\alpha}$.

We have reached a contradiction, so this case holds.

$\Leftarrow$ Switch HP and TH from the point above.

Analgously, we proceed by contradiction:

- $\forall \mathsf{A}, \overline{\alpha}.\ \vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}$ and $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \Omega$ and $\mathtt{heaps}(\overline{\alpha}) \not\approx_\beta \mathtt{heaps}(\overline{\alpha})$

By the same reasoning as above, with the HP we have we obtain $\mathsf{M} \vdash \overline{\alpha}$ and $\mathbf{M} \vdash \overline{\alpha}$.

Again by 3.b in Definition 8 ($\mathsf{M}\mathcal{R}\mathbf{M}$) we know that the heaps of all actions in the traces are related.

Therefore, $\mathtt{heaps}(\overline{\alpha}) \approx_\beta \mathtt{heaps}(\overline{\alpha})$, which gives us a contradiction.

□

12.2 Proof of Theorem 2 (Compiler $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ is *CC*)

Proof. The proof proceeds for $\beta_0 = (\ell, \mathbf{0}, \mathbf{k_{root}})$ and then, given that the languages are deterministic, by Lemma 8 (Generalised compiler correctness for $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$) as initial states are related by definition. □

♠

Lemma 7 (Expressions compiled with $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ are related)**.**

$$\begin{array}{rl} & \forall \\ \text{if} & \mathsf{H} \approx_\beta \mathbf{H} \\ & \mathsf{H} \triangleright \mathsf{e}\rho \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v} \\ \text{then} & \mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \end{array}$$

Proof. This proof proceeds by structural induction on e.

Base case: Values

true By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-True), $[\![\mathsf{true}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{0}$.
As $\mathsf{true} \approx_\beta \mathbf{0}$, this case holds.

false Analogous to the first case by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-False).

$\mathsf{n} \in \mathbb{N}$ Analogous to the first case by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-nat).

x Analogous to the first case, by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Var) and by the relatedness of the substitutions.

ℓ Analogous to the first case by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Loc).

$\langle \mathsf{v}, \mathsf{v} \rangle$ By induction on v by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Pair) and then it is analogous to the first case.

Inductive case: Expressions

$\mathsf{e} \oplus \mathsf{e}'$ By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-op) we have that
$[\![\mathsf{e} \oplus \mathsf{e}']\!]^{\mathsf{L^U}}_{\mathbf{L^P}} = [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \oplus [\![\mathsf{e}']\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$
By HP we have that $\mathsf{H} \triangleright \mathsf{e}\rho \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{n}$ and $\mathsf{H} \triangleright \mathsf{e}'\rho \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{n}'$.
By Rule $\mathsf{EL^U}$-op we have that $\mathsf{H} \triangleright \mathsf{n} \oplus \mathsf{n}' \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{n}''$.
By IH we have that $\mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\mathsf{n}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ and $\mathbf{H} \triangleright [\![\mathsf{e}']\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\mathsf{n}']\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$.
By Rule $\mathbf{EL^P}$-op we have that $\mathbf{H} \triangleright [\![\mathsf{n}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \oplus [\![\mathsf{n}']\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\mathsf{n}'']\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$.
So this case holds.

$\mathsf{e} \otimes \mathsf{e}'$ Analogous to the case above by IH, Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-cmp), Rule $\mathsf{EL^U}$-comp and Rules $\mathbf{EL^P}$-op and $\mathbf{EL^P}$-if-true.

$!\mathsf{e}$ Analogous to the case above by IH twice, Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Deref), Rule EL$^{\mathsf{U}}$-dereference and Rules EL$^{\mathbf{P}}$-p1, EL$^{\mathbf{P}}$-p2 and EL$^{\mathbf{P}}$-letin and a case analysis by Rules EL$^{\mathbf{P}}$-deref-top and EL$^{\mathbf{P}}$-deref-k.

$\langle \mathsf{e}, \mathsf{e} \rangle$ Analogous to the case above by IH and Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Pair).

$\mathsf{e.1}$ By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-P1) $[\![\mathsf{e.1}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} = [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.1}$.

By HP $\mathsf{H} \triangleright \mathsf{e.1}\rho \hookrightarrow\!\!\!\!\rightarrow \langle \mathsf{v_1}, \mathsf{v_2} \rangle \hookrightarrow\!\!\!\!\rightarrow \mathsf{v_1}$.

By IH we have that $\mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.1}[\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\rightarrow [\![\langle \mathsf{v_1}, \mathsf{v_2} \rangle]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.1}$.

By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Pair) we have that $[\![\langle \mathsf{v_1}, \mathsf{v_2} \rangle]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\mathbf{.1} = \left\langle [\![\mathsf{v_1}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}, [\![\mathsf{v_2}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \right\rangle \mathbf{.1}$.

Now $\mathbf{H} \triangleright \left\langle [\![\mathsf{v_1}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}, [\![\mathsf{v_2}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \right\rangle \mathbf{.1} \hookrightarrow\!\!\!\!\rightarrow [\![\mathsf{v_1}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$.

So this case holds.

$\mathsf{e.2}$ Analogous to the case above by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-P2), Rule EL$^{\mathsf{U}}$-p2 and Rule EL$^{\mathbf{P}}$-p2.

□

♠

Lemma 8 (Generalised compiler correctness for $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$).

Proof.

$$
\begin{aligned}
&\forall \ldots \exists \beta' \\
\text{if } &\vdash \mathsf{C} : \mathsf{whole} \\
&\mathsf{C} = \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}} \\
&[\![\mathsf{C}]\!]^{\mathsf{L^\tau}}_{\mathbf{L^\pi}} = \mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}} = \mathbf{C} \\
&\mathsf{C}, \mathsf{H} \triangleright \mathsf{s} \approx_\beta \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{s}]\!]^{\mathsf{L^\tau}}_{\mathbf{L^\pi}} \\
&\mathsf{C}, \mathsf{H} \triangleright \mathsf{s}\rho \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \triangleright \mathsf{s}'\rho' \\
\text{then } &\mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{s}]\!]^{\mathsf{L^\tau}}_{\mathbf{L^\pi}}[\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \xrightarrow{\lambda} \mathbf{C}', \mathbf{H}' \triangleright [\![\mathsf{s}']\!]^{\mathsf{L^\tau}}_{\mathbf{L^\pi}}[\![\rho']\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \\
&\mathbf{C}' = \mathbf{k_{root}}; \overline{\mathbf{F}}; \overline{\mathbf{I}} \\
&\mathsf{C}, \mathsf{H} \triangleright \mathsf{s}'\rho' \approx_{\beta'} \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{s}']\!]^{\mathsf{L^\tau}}_{\mathbf{L^\pi}}[\![\rho']\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \\
&\beta \subseteq \beta'
\end{aligned}
$$

The proof proceeds by induction on C and the on the reduction steps.

Base case

skip By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Skip) this case follows trivially.

Inductive

$\mathsf{let}\ \mathsf{x} = \mathsf{new}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}$

By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-New) $[\![\mathsf{let}\ \mathsf{x} = \mathsf{new}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} =$

$$\begin{aligned}
&\mathbf{let}\ \mathbf{x_{loc}} = \mathbf{new}\ [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{in}\\
&\mathbf{let}\ \mathbf{x_{cap}} = \mathbf{hide}\ \mathbf{x_{loc}}\ \mathbf{in}\\
&\mathbf{let}\ \mathbf{x} = \langle \mathbf{x_{loc}}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}
\end{aligned}$$

By HP $\mathsf{H} \triangleright \mathsf{e}\rho \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}$

So by Lemma 7 we have HPE: $\mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\rightarrow [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ and HPV $\mathsf{v} \approx_\beta [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$.

By Rule $\mathsf{EL^U}$-alloc: $\mathsf{C}; \mathsf{H} \triangleright \mathsf{let}\ \mathsf{x} = \mathsf{new}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H}; \ell \mapsto \mathsf{v} \triangleright \mathsf{s}[\ell\ /\ \mathsf{x}]$.

So by HPE:

$$\begin{aligned}
&\mathbf{C}; \mathbf{H} \triangleright \mathbf{let}\ \mathbf{x_{loc}} = \mathbf{new}\ [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\ \mathbf{in}\\
&\qquad \mathbf{let}\ \mathbf{x_{cap}} = \mathbf{hide}\ \mathbf{x_{loc}}\ \mathbf{in}\\
&\qquad \mathbf{let}\ \mathbf{x} = \langle \mathbf{x_{loc}}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\rho\\
&\text{Rule } \mathbf{EL^P}\text{-new}\\
&\xrightarrow{\epsilon} \mathbf{C}; \mathbf{H}; \mathbf{n} \mapsto [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \bot \triangleright\ \mathbf{let}\ \mathbf{x_{cap}} = \mathbf{hide}\ \mathbf{x_{loc}}\ \mathbf{in}\\
&\qquad \mathbf{let}\ \mathbf{x} = \langle \mathbf{x_{loc}}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\mathbf{n}\ /\ \mathbf{x_{loc}}]\\
&\equiv \mathbf{C}; \mathbf{H}; \mathbf{n} \mapsto [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \bot \triangleright\ \mathbf{let}\ \mathbf{x_{cap}} = \mathbf{hide}\ \mathbf{n}\ \mathbf{in}\\
&\qquad \mathbf{let}\ \mathbf{x} = \langle \mathbf{n}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\\
&\text{Rule } \mathbf{EL^P}\text{-hide}\\
&\xrightarrow{\epsilon} \mathbf{C}; \mathbf{H}; \mathbf{n} \mapsto [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \mathbf{k}; \mathbf{k} \triangleright\ \mathbf{let}\ \mathbf{x} = \langle \mathbf{n}, \mathbf{x_{cap}} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\mathbf{k}\ /\ \mathbf{x_{cap}}]\\
&\equiv \mathbf{C}; \mathbf{H}; \mathbf{n} \mapsto [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \mathbf{k}; \mathbf{k} \triangleright\ \mathbf{let}\ \mathbf{x} = \langle \mathbf{n}, \mathbf{k} \rangle\ \mathbf{in}\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\rho\\
&\text{Rule } \mathbf{EL^P}\text{-letin}\\
&\xrightarrow{\epsilon} \mathbf{C}; \mathbf{H}; \mathbf{n} \mapsto [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \mathbf{k}; \mathbf{k} \triangleright\ [\![\mathsf{s}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\langle \mathbf{n}, \mathbf{k} \rangle\ /\ \mathbf{x}]
\end{aligned}$$

Let $\beta' = \beta \cup (\ell, \mathbf{n}, \mathbf{k})$.

By definition of $\approx_{\beta'}$ and by β' we get HPL $\ell \approx_{\beta'} \langle \mathbf{n}, \mathbf{k} \rangle$.

By a simple weakening lemma for β for substitutions and values applied to HP and HPV we can get HPVB $\mathsf{v} \approx_{\beta'} [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$.

As $\mathsf{H} \approx_\beta \mathbf{H}$ by HP, by a simple weakening lemma get that $\mathsf{H} \approx_{\beta'} \mathbf{H}$ too and by Rule Heap relation with HPL and HPVB we get $\mathsf{H}' \approx_{\beta'} \mathbf{H}'$.

We have that $\rho' = \rho[\ell\ /\ \mathsf{x}]$ and $\boldsymbol{\rho}' = [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\langle \mathbf{n}, \mathbf{k} \rangle\ /\ \mathbf{x}]$.

So by HPL we get that $\rho' \approx_{\beta'} \boldsymbol{\rho}'$.

$\mathsf{s; s'}$ Analogous to the case above by IH, Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Seq) and a case analysis on what s reduces to, either with Rule $\mathsf{EL^U}$-sequence and Rule $\mathbf{EL^P}$-sequence or with Rule $\mathsf{EL^U}$-step and Rule $\mathbf{EL^P}$-step.

$\mathsf{let\ x = e\ in\ s}$ Analogous to the case above by IH, Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Letin), Rule $\mathsf{EL^U}$-letin and Rule $\mathbf{EL^P}$-letin.

$\mathsf{x := e'}$ Analogous to the case above by Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Assign), Rule $\mathsf{EL^U}$-update and Rule $\mathbf{EL^P}$-letin (twice), Rules $\mathbf{EL^P}$-p1 and $\mathbf{EL^P}$-p2 and then a case analysis by Rules $\mathbf{EL^P}$-assign-top and $\mathbf{EL^P}$-assign-k.

$\mathsf{if\ e\ then\ s\ else\ s'}$ Analogous to the case above by IH, Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-If) and then either Rule $\mathsf{EL^U}$-if-true and Rule $\mathbf{EL^P}$-if-true or Rule $\mathsf{EL^U}$-if-false and Rule $\mathbf{EL^P}$-if-false.

$\mathsf{call\ f\ e}$ By Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-call) $[\![\mathsf{call\ f\ e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} = \mathbf{call\ f}\ [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$

By HP $\mathsf{H} \triangleright \mathsf{e}\rho \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}$

So by Lemma 7 we have HPE: $\mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\rho \hookrightarrow\!\!\!\!\rightarrow [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ and HPR $\mathsf{v} \approx_\beta [\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$.

So as C is whole, we apply Rule $\mathsf{EL^U}$-call-internal

$$\begin{aligned}&\mathsf{C, H} \triangleright (\mathsf{call\ f\ e}\rho)_{\overline{\mathsf{f'}}} \xrightarrow{\epsilon}\\ &\mathsf{C, H} \triangleright (\mathsf{s; return;}\rho[\mathsf{v} \;/\; \mathsf{x}])_{\overline{\mathsf{f'}};\mathsf{f}}\end{aligned}$$

By Rule $\mathbf{EL^P}$-call-internal

$$\begin{aligned}&\mathbf{C, H} \triangleright \left(\mathbf{call\ f}\ [\![\mathsf{e}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right)_{\overline{\mathbf{f'}}} \xrightarrow{\epsilon}\\ &\mathbf{C, H} \triangleright \left(\mathbf{s; return;}[\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\left[[\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \;\middle/\; \mathbf{x}\right]\right)_{\overline{\mathbf{f'}};\mathbf{f}}\end{aligned}$$

By the first induction on C we get

IH1 $\mathsf{C, H} \triangleright (\mathsf{s; return;}\rho')_{\overline{\mathsf{f'}};\mathsf{f}} \approx_\beta \mathbf{C, H} \triangleright (\mathbf{s; return;}\rho')_{\overline{\mathbf{f'}};\mathbf{f}}$

We instantiate ρ' with $\rho[\mathsf{v} \;/\; \mathsf{x}]$ and ρ' with $[\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\left[[\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \;\middle/\; \mathbf{x}\right]$.

So by HP and HPR we have that $\rho[\mathsf{v} \;/\; \mathsf{x}] \approx_\beta [\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\left[[\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \;\middle/\; \mathbf{x}\right]$

We we can use IH1 to conclude

$\mathsf{C, H} \triangleright (\mathsf{s; return;}\rho[\mathsf{v} \;/\; \mathsf{x}])_{\overline{\mathsf{f'}};\mathsf{f}} \approx_\beta \mathbf{C, H} \triangleright \left(\mathbf{s; return;}[\![\rho]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\left[[\![\mathsf{v}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} \;\middle/\; \mathbf{x}\right]\right)_{\overline{\mathbf{f'}};\mathbf{f}}$

As $\beta' = \beta$, this case holds.

□

12.3 Proof of Theorem 3 (Compiler $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ is *RSC*)

Proof. HPM: $\mathsf{M} \approx_\beta \mathbf{M}$

HP1: $\mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$

TH1: $\mathbf{M} \vdash [\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \mathbf{rs}$

We can state it in contrapositive form as:

HP2: $\mathbf{M} \nvdash [\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}} : \mathbf{rs}$

TH2: $\mathsf{M} \nvdash \mathsf{C} : \mathsf{rs}$

By expanding the definition of *rs* in HP2 and TH2, we get

HP21 $\exists \mathbf{A}, \overline{\alpha}.\mathbf{M} \vdash \mathbf{A} : \mathbf{attacker}$ and either $\nvdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right] : \mathbf{whole}$ or HPRT1 $(\mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right]\right) \xRightarrow{\overline{\alpha}} \_$ and HPRMT1 $\mathbf{M} \nvdash \overline{\alpha}$)

TH21 $\exists \mathsf{A}, \overline{\alpha}.\mathsf{M} \vdash \mathsf{A} : \mathsf{attacker}$ and either $\nvdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}$ or TH2 $(\mathsf{\Omega_0}\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \_$ and TH4 $\mathsf{M} \nvdash \overline{\alpha}$)

We consider the case of a whole $\mathbf{A}$, the other is trivial.

We can apply Theorem 4 ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ is correct) with HPRT1 and instantiate A with a $\mathbf{A}$ from $\langle\!\langle\mathbf{A}\rangle\!\rangle$ and we get the following unfolded HPs

HPRS $\mathsf{\Omega_0}\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \mathsf{\Omega}$

HPRel $\overline{\alpha} \approx_\beta \overline{\alpha}$.

So TH3 holds by HPRS.

We need to show TH4

Assume by contradiction HPBOT: the monitor in the source does not fail: $\mathsf{M} \vdash \overline{\alpha}$)

By Rule $\mathsf{L^U}$-valid trace we know that forall $\alpha \in \overline{\alpha}$ such that $\texttt{heaps}(\alpha) = \mathsf{H}$, this holds: HPHR $\mathsf{M}; \mathsf{H} \rightsquigarrow \mathsf{M}'$.

We can expand HPHR by Rule $\mathsf{L^U}$-Monitor Step and get:

HPMR: $(\sigma_{\mathsf{c}}, \mathsf{H}'_{\mathsf{h}}, \sigma_{\mathsf{f}}) \in \rightsquigarrow$

for a heap $\mathsf{H}'_{\mathsf{h}} \subseteq \mathsf{H}_{\mathsf{h}}$

By HPM $\mathsf{M} \approx_\beta \mathbf{M}$ for initial states.

By Definition 9 ($\mathsf{M} \approx \mathbf{M}$) and the second clause of Definition 8 ($\mathsf{M}\mathcal{R}\mathbf{M}$) with HPMR we know that $\mathsf{M} \approx_\beta \mathbf{M}$ for the current states.

By the first clause of Definition 8 ($\mathsf{M}\mathcal{R}\mathbf{M}$) we know that

HPMRBI: $(\sigma_{\mathsf{c}}, \mathsf{H}, \_) \in \rightsquigarrow \iff (\sigma_{\mathbf{c}}, \mathbf{H}, \_) \in \rightsquigarrow$

By HPMRBI with HPMR we know that

HPMRTC: $(\sigma_{\mathbf{c}}, \mathbf{H}'_{\mathbf{h}}, \sigma_{\mathbf{f}}) \in \rightsquigarrow$

However, by HPRMT1 and Rule $\mathbf{L^P}$-valid trace we know that

HPNR: $\mathbf{M}; \mathbf{H} \not\rightsquigarrow$

so we get

HPCON: $\nexists(\sigma_{\mathbf{c}}, \mathbf{H}'_{\mathbf{h}}, \sigma_{\mathbf{f}}) \in \rightsquigarrow$

By HPCON and HPMRTC we get the contradiction, so the proof holds. □

♠

12.4 Proof of Lemma 1 (Compiled code steps imply existence of source steps)

Proof. The proof proceeds by induction on $\xRightarrow{\alpha!}$.

Base case: $\xRightarrow{\alpha!}$

By Rule $\mathbf{EL^{P}}$-single we need to prove the silent steps and the $\alpha!$ action.

ϵ

The proof proceeds by analysis of the target reductions.

Rule $\mathbf{EL^{P}}$-sequence In this case we do not need to pick and the thesis holds by Rule $\mathsf{EL^{U}}$-sequence.

Rule $\mathbf{EL^{P}}$-step In this case we do not need to pick and the thesis holds by Rule $\mathsf{EL^{U}}$-step.

Rule $\mathbf{EL^{P}}$-if-true We have: $\mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L^{U}}}_{\mathbf{L^{P}}} \rho \hookrightarrow\!\!\!\!\to \mathbf{0}$

We apply Lemma 9 (Compiled code expression steps implies existence of source expression steps) and obtain a $\mathsf{v} \approx_\beta \mathbf{0}$

By definition we have $\mathsf{0} \approx_\beta \mathbf{0}$ and $\mathsf{true} \approx_\beta \mathbf{0}$, we pick the second.

So we have $\mathsf{H} \triangleright \mathsf{e}\rho \hookrightarrow\!\!\!\!\to \mathsf{true}$

We can now apply Rule $\mathsf{EL^{U}}$-if-true and this case follows.

Rule $\mathbf{EL^{P}}$-if-false This is analogous to the case above.

Rule $\mathbf{EL^{P}}$-assign-top Analogous to the case above.

Rule $\mathbf{EL^{P}}$-assign-k This is analogous to the case above but for $\mathsf{v} = \ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle$.

Rule $\mathbf{EL^{P}}$-letin This follows by Lemma 9 and by Rule $\mathsf{EL^{U}}$-letin.

Rule $\mathbf{EL^{P}}$-new This follows by Lemma 9 and by Rule $\mathsf{EL^{U}}$-alloc.

Rule $\mathbf{EL^{P}}$-hide By analisis of compiled code we know this only happens after a **new** is executed.

In this case we do not need to perform a step in the source and the thesis holds.

Rule $\mathbf{EL^{P}}$-call-internal This follows by Lemma 9 and by Rule $\mathsf{EL^{U}}$-call-internal.

Rule $\mathbf{EL^{P}}$-ret-internal In this case we do not need to pick and the thesis holds by Rule $\mathsf{EL^{U}}$-ret-internal.

$\alpha!$

The proof proceeds by case analysis on $\alpha!$

`call` $\mathbf{f}$ $\mathbf{v}$ $\mathbf{H}!$ This follows by Lemma 9 (Compiled code expression steps implies existence of source expression steps) and by Rule $\mathsf{EL^{U}}$-callback.

`ret` $\mathbf{H}!$ In this case we do not need to pick and the thesis holds by Rule $\mathsf{EL^{U}}$-return.

Inductive case: This follows from IH and the same reasoning as for the single action above.

□

♠

Lemma 9 (Compiled code expression steps implies existence of source expression steps)**.**

$$
\begin{aligned}
& \forall \\
\text{if } \; & \mathbf{H} \triangleright \llbracket e \rrbracket^{L^U}_{\mathbf{L^P}} \rho \hookrightarrow\!\!\!\!\rightarrow \mathbf{v} \\
\text{and if } \; & \{\rho\} = \{\rho \mid \rho \approx_\beta \rho\} \\
& v \approx_\beta \mathbf{v} \\
& H \approx_\beta \mathbf{H} \\
\text{then } \; & \exists \rho_j \in \{\rho\} \, . \, H \triangleright e\rho_j \hookrightarrow\!\!\!\!\rightarrow v
\end{aligned}
$$

Proof. This proceeds by structural induction on e.

Base case: $true$ This follows from Rule ($\llbracket\cdot\rrbracket^{L^U}_{\mathbf{L^P}}$-True).

$false$ This follows from Rule ($\llbracket\cdot\rrbracket^{L^U}_{\mathbf{L^P}}$-False).

$n \in \mathbb{N}$ This follows from Rule ($\llbracket\cdot\rrbracket^{L^U}_{\mathbf{L^P}}$-nat).

x This follows from the relation of the substitutions and the totality of $\approx_\beta$ and Rule ($\llbracket\cdot\rrbracket^{L^U}_{\mathbf{L^P}}$-Var).

$\langle v, v' \rangle$ This follows from induction on v and v'.

Inductive case: $e \oplus e'$ By definition of $\approx_\beta$ we know that v and v' could be either natural numbers or booleans.
We apply the IH with:
IHV1 $n \approx_\beta \mathbf{n}$
IHV2 $n' \approx_\beta \mathbf{n}'$
By IH we get
IHTE1 $\mathbf{H} \triangleright \llbracket e \rrbracket^{L^U}_{\mathbf{L^P}} \rho \hookrightarrow\!\!\!\!\rightarrow \mathbf{n}$
IHTE2 $\mathbf{H} \triangleright \llbracket e' \rrbracket^{L^U}_{\mathbf{L^P}} \rho \hookrightarrow\!\!\!\!\rightarrow \mathbf{n}'$
IHSE1 $H \triangleright e\rho_j \hookrightarrow\!\!\!\!\rightarrow n$
IHSE2 $H \triangleright e'\rho_j \hookrightarrow\!\!\!\!\rightarrow n'$
By Rule ($\llbracket\cdot\rrbracket^{L^U}_{\mathbf{L^P}}$-op) we have that $\llbracket e \oplus e' \rrbracket^{L^U}_{\mathbf{L^P}} = \llbracket e \rrbracket^{L^U}_{\mathbf{L^P}} \oplus \llbracket e' \rrbracket^{L^U}_{\mathbf{L^P}}$.
By Rule $\mathbf{EL^P}$-op with IHTE1 and IHTE2 we have that $\mathbf{H} \triangleright \llbracket e \rrbracket^{L^U}_{\mathbf{L^P}} \oplus \llbracket e' \rrbracket^{L^U}_{\mathbf{L^P}} \hookrightarrow\!\!\!\!\rightarrow \mathbf{n}''$
where IHVT $n'' = n \oplus n'$
By Rule EL^U-op with IHSE1 and IHSE2 we have that $H \triangleright e \oplus e' \hookrightarrow\!\!\!\!\rightarrow n''$
if $n'' = n \oplus n'$
This follows from IHVT and IHV1 and IHV2.

$\mathsf{e} \otimes \mathsf{e}'$ As above, this follows from IH and Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-cmp) and Rule $\mathsf{EL^U}$-comp.

$\langle \mathsf{e}, \mathsf{e}' \rangle$ As above, this follows from IH and Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Pair).

$\mathsf{e.1}$ As above, this follows from IH and Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-P1) and Rule $\mathsf{EL^U}$-p1.

$\mathsf{e.2}$ Analogous to the case above.

$!\mathsf{e}$ As above, this follows from IH and Rule ($[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$-Deref) and Rule $\mathsf{EL^U}$-dereference but with the hypothesis that e evaluates to a v related to a $\langle \mathbf{n}, \mathbf{v} \rangle$.

□

♠

12.5 Proof of Theorem 4 ($\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$ is correct)

Proof. HP1 $\mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right]\right) \xRightarrow{\overline{\alpha}} \mathbf{\Omega}$

HPF $\mathbf{\Omega} \xRightarrow{\epsilon} \mathbf{\Omega}'$

HPN $\overline{\mathbf{I}} = \mathtt{names}(\mathbf{A})$
HPT $\overline{\alpha} \equiv \overline{\alpha'} \cdot \alpha?$
HPL $\ell_{\mathsf{i}}; \ell_{\mathsf{glob}} \notin \beta$
THE $\exists \mathsf{A} \in \langle\!\langle \mathbf{I}, \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$
TH1 $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \Omega$
THA $\overline{\alpha} \approx_{\beta} \overline{\alpha}$
THS $\Omega \approx_{\beta} \mathbf{\Omega}$
THC $\Omega.\mathsf{H}.\ell_{\mathsf{i}} = \|\overline{\alpha}\| + 1$
The proof proceeds by induction on $\overline{\alpha'}$.

Base case: We perform a case analysis on $\alpha?$

$\mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?$

Given
HP1 $\mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right]\right) \xRightarrow{\mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?} \mathbf{\Omega}$
We need to show that
THE $\exists \mathsf{A} \in \langle\!\langle \mathbf{I}, \overline{\alpha} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$
TH1 $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \Omega$
THA $\mathtt{call}\ \mathsf{f}\ \mathsf{v}\ \mathsf{H}? \approx_{\beta} \mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?$
THS $\Omega \approx_{\beta} \mathbf{\Omega}$
THC $\Omega.\mathsf{H}.\ell_{\mathsf{i}} = \|\overline{\alpha}\| + 1$

By Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-call) the back-translated context executes this code inside main:

```
if $!\ell_i == n$ then
  incrementCounter()
  let x1 = new $v_1$ in register($\langle x1, n_1\rangle$)
  ...
  let xj = new $v_j$ in register($\langle xj, n_j\rangle$)
  call f v
 else skip
```

As $\mathbf{H_{pre}}$ is $\varnothing$, no updates are added.

Given that ℓ_i is initialised to 1 in Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-skel), this code is executed and it generates action $\mathtt{call}\ \mathsf{f}\ \mathsf{v}\ \mathsf{H}?$ where $\mathsf{H}{=}\ell_1 \mapsto \mathsf{v}_1; \cdots ; \ell_n \mapsto \mathsf{v}_n$ for all $\mathbf{n_i} \in \mathtt{dom}(\mathbf{H})$ such that $\ell_i \approx_\beta \langle \mathbf{n_i}, \_\rangle$ and:

HPHR $\mathsf{H} \approx_\beta \mathbf{H}$

By HPHR, Lemma 11 (Backtranslated values are related) and Lemma 1 (Compiled code steps imply existence of source steps) with HPF we get THA, THE and TH1.

By Rule Related states – Secure, THS holds too.

Execution of incrementCounter() satisfies THC.

$\mathtt{ret}\ \mathbf{H}?$ This cannot happen as by Rule $\mathbf{EL^P}$-retback there needs to be a running process with a non-empty stack and by Rule $\mathbf{L^P}$-Initial State the stack of initial states is empty and the only way to add to the stack is performing a call via Rule $\mathbf{EL^P}$-call, which would be a different label.

Inductive case:

We know that (eliding conditions HP that are trivially satisfied):

IHP1 $\mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right]\right) \xRightarrow{\overline{\alpha}} \mathbf{\Omega'} \xRightarrow{\alpha!} \mathbf{\Omega''} \xRightarrow{\alpha?} \mathbf{\Omega}$

And we need to prove:

ITH1 $\Omega_0\left(\langle\!\langle \mathbf{I}, \overline{\alpha}\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}[\mathsf{C}]\right) \xRightarrow{\overline{\alpha}} \Omega' \xRightarrow{\alpha!} \Omega'' \xRightarrow{\alpha?} \Omega$

ITHA $\overline{\alpha}\alpha!\alpha? \approx_\beta \overline{\alpha}\alpha!\alpha?$

ITHS $\Omega \approx_\beta \mathbf{\Omega}$

And the inductive HP is (for $\varnothing \subseteq \beta'$):

IH-HP1 $\mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}\right]\right) \xRightarrow{\overline{\alpha}} \mathbf{\Omega'}$

IH-TH1 $\Omega_0\left(\langle\!\langle \mathbf{I}, \overline{\alpha}\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}[\mathsf{C}]\right) \xRightarrow{\overline{\alpha}} \Omega'$

IH-THA $\overline{\alpha} \approx_{\beta'} \overline{\alpha}$

IH-THS $\Omega' \approx_{\beta'} \mathbf{\Omega}'$

By IHP1 and HPF we can apply Lemma 1 (Compiled code steps imply existence of source steps) and so we can apply the IH to get IH-TH1, IH-THA and IH-THS.

We perform a case analysis on $\alpha!$, and show that the back-translated code performs $\alpha!$.

By IH we have that the existing code is generated by Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-listact-i): $\langle\!\langle\overline{\alpha}, n, \mathbf{H_{pre}}, \mathbf{ak}, \overline{\mathsf{f}}\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$.

The next action $\alpha!$ produces code according to:

HPF $\langle\!\langle\alpha!, n, \mathbf{H_{pre}}, \mathbf{ak}, \overline{\mathsf{f}}\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$.

By Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-join), code of this action is the first if statement executed.

`call` **f v H**! By Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-callback-loc) this code is placed at function f so it is executed when compiled code jumps there

```
if $!\ell_i == n$ then
  incrementCounter()
  let l1 = $e_1$ in register($\langle$l1, $n_1\rangle$)
  ...
  let lj = $e_j$ in register($\langle$lj, $n_j\rangle$)
 else skip
```

By IH we have that $\ell_i \mapsto \mathsf{n}$, so we get

IHL $\ell_i \mapsto \mathsf{n}+1$

By Definition 15 (Reachable) we have for $i \in 1..j$ that a reachable location $\mathbf{n_i} \in \mathtt{dom}(\mathbf{H})$ has a related counterpart in $\ell_i \in \mathtt{dom}(\mathsf{H})$ such that $\mathsf{H} \triangleright \mathsf{e_i} \hookrightarrow\!\!\!\rightarrow \ell_i$.

By Lemma 10 (L^τ attacker always has access to all capabilities) we know all capabilities to access any $\mathbf{n_i}$ are in **ak**.

We use **ak** to get the right increment of the reach.

`ret` **H**! In this case from IHF we know that $\overline{\mathsf{f}} = \mathsf{f}'\overline{\mathsf{f}'}$.

This code is placed at f', so we identify the last called function and the code is placed there. Source code returns to f' so this code is executed Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-ret-loc)

```
if $!\ell_i == n$ then
  incrementCounter()
  let l1 = $e_1$ in register($\langle$l1, $n_1\rangle$)
  ...
  let lj = $e_j$ in register($\langle$lj, $n_j\rangle$)
 else skip
```

This case now follows the same reasoning as the one above.

So we get (for $\beta' \subseteq \beta''$):

HP-AC! $\alpha! \approx_{\beta''} \alpha!$

By IH-THS and Rule Related states – Whole and HP-AC! we get HP-OM2:

HP-OM2: $\Omega'' \approx_{\beta''} \mathbf{\Omega''}$

The next action $\alpha?$ produces code according to:

IHF1 $\langle\!\langle \alpha?, n+1, \mathbf{H'_{pre}}, \mathbf{ak'}, \overline{\mathsf{f'}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$.

We perform a case analysis on $\alpha?$ and show that the back-translated code performs $\alpha?$:

`ret` $\mathbf{H}?$ By Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-retback), after n actions, we have from IHF1 that $\overline{\mathsf{f'}} = \mathsf{f'}\overline{\mathsf{f''}}$ and inside function $\mathsf{f'}$ there is this code:

```
if !ℓ_i == n then
  let x1 = new v_1 in register(⟨x1, n_1⟩)
  ...
  let xj = new v_j in register(⟨xj, n_j⟩)
  update(m_1, u_1)
  ...
  update(m_l, u_l)
 else skip
```

By IHL, $\ell_i \mapsto \mathsf{n}+1$, so the if gets executed.

By definition, forall $\mathbf{n} \in \mathtt{dom}(\mathbf{H})$ we have that $\mathbf{n} \in \mathbf{H_n}$ or $\mathbf{n} \in \mathbf{H_c}$ (from the case definition).

By Lemma 10 (L^τ attacker always has access to all capabilities) we know all capabilities to access any $\mathbf{n}$ are in $\mathbf{ak}$.

We induce on the size of $\mathbf{H}$; the base case is trivial and the inductive case follows from IH and the following:

$\mathbf{H_n}$**:** and $\mathbf{n}$ is newly allocated.

In this case when we execute

$\mathsf{C}; \mathsf{H'} \triangleright \mathsf{let}\ \mathsf{x1} = \mathsf{new}\ \mathsf{v_1}\ \mathsf{in}\ \mathsf{register}(\langle \mathsf{x1}, \mathsf{n_1}\rangle) \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H'}; \ell'' \mapsto \langle\!\langle \mathbf{v_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \triangleright \mathsf{register}(\langle \ell'', \mathsf{n_1}\rangle)$

and we create β'' by adding $\ell'', \mathbf{n}, \eta'$ to β'.

By Lemma 2 ($\mathsf{register}(\ell, \mathsf{n})$ does not add duplicates for n) we have that:

$\mathsf{C}; \mathsf{H'}; \ell'' \mapsto \langle\!\langle \mathbf{v_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \triangleright \mathsf{register}(\langle \ell'', \mathsf{n_1}\rangle) \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H'}; \ell'' \mapsto \langle\!\langle \mathbf{v_1} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} \triangleright \mathsf{skip}$

and we can lookup ℓ'' via n.

$\mathbf{H_c}$**:** and $\mathbf{n}$ is already allocated.

In this case

$\mathsf{C}; \mathsf{H'} \triangleright \mathsf{update}(\mathsf{m_1}, \mathsf{u_1}) \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H''} \triangleright \mathsf{skip}$

By Lemma 3 ($\mathsf{update}(\mathsf{n},\mathsf{v})$ never gets stuck) we know that $\mathsf{H}'' = \mathsf{H}'[\ell'' \mapsto \_ \ / \ \ell'' \mapsto \mathsf{u}_1]$
and ℓ'' such that $(\ell'', \mathbf{m_1}, \eta'') \in \beta'$.

By Lemma 11 (Backtranslated values are related) on the values stored on the heap, let the heap after these reduction steps be H, we can conclude

HPRH $\mathsf{H} \approx_{\beta''} \mathbf{H}$.

As no other if inside f is executed, eventually we hit its return statement, which by Rule ($\langle\!\langle\cdot\rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}$ -join) and Rule ($\langle\!\langle\cdot\rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}$ -fun) is $\mathsf{incrementCounter}(); \mathsf{return};$.

Execution of $\mathsf{incrementCounter}()$ satisfies THC.

So we have $\Omega'' \xRightarrow{\texttt{ret H?}} \Omega$ (by Lemma 11) and with HPRH.

$\texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?$ Similar to the base case, only with update bits, which follow the same reasoning above.

So we get (for $\beta'' \subseteq \beta$):

HP-AC? $\alpha? \approx_\beta \alpha?$

By IH-THA and HP-AC! and HP-AC? we get ITHA.

Now by Rule Related states – Whole again and HP-AC? we get ITHS and ITH1, so the theorem holds.

□

♠

Lemma 10 (L^τ attacker always has access to all capabilities)**.**

$$\begin{aligned}
&\forall\\
&\text{if } \langle\!\langle \alpha, n, \mathbf{H}, \mathbf{ak}, \overline{\mathsf{f}} \rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}} = \{\mathsf{s}, \mathbf{ak}', \mathbf{H}', \overline{\mathsf{f}'}, \mathsf{f}\}\\
&\mathbf{k}.\,\mathbf{n} \mapsto \mathbf{v} : \mathbf{k} \in \mathbf{H}\\
&\text{then } \mathbf{n} \in \texttt{reach}(\mathbf{ak}'.\texttt{loc}, \mathbf{ak}'.\texttt{cap}, \mathbf{H})
\end{aligned}$$

Proof. Trivial case analyisis on Rules ($\langle\!\langle\cdot\rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}$ -ret-loc) to ($\langle\!\langle\cdot\rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}$ -callback-loc). □

♠

Lemma 11 (Backtranslated values are related)**.**

$$\begin{aligned}
&\forall \mathbf{v}, \beta.\\
&\langle\!\langle \mathbf{v} \rangle\!\rangle_{\mathsf{L^U}}^{\mathbf{L^P}} \approx_\beta \mathbf{v}
\end{aligned}$$

Proof. Trivial analysis of Section 4.2.1. □

♠

12.6 Proof of Theorem 5 (Typability Implies Robust Safety in L^{τ})

Proof. HP1 $\vdash \mathsf{C} : \mathsf{UN}$

HP2 $\mathsf{C} \frown \mathsf{M}$

TH $\mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$

We expand TH: $\forall \mathsf{A}, \overline{\alpha}.\mathsf{M} \vdash \mathsf{A} : \mathsf{attacker}$ and $\vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}$ if HPR $\Omega_0 \left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \_$ then THM $\mathsf{M} \vdash \overline{\alpha}$

By definition of `heaps`($\overline{\alpha}$) and by Rule L^{τ}-valid trace we get a $\overline{\mathsf{H}}$ to induce on.

The base case holds by Rule L^{τ}-Monitor Step Trace Base.

In the inductive case we are considering $\overline{\mathsf{H}} \cdot \mathsf{H}$ and the IH covers the first part of the trace.

By Lemma 12 (L^{τ}-α reductions respect heap typing), given that the state generating the action is $\mathsf{C}, \mathsf{H} \triangleright \Pi$ we know that, HPH $\vdash \mathtt{mon\text{-}care}(\mathsf{H}, \Delta) : \Delta$

By Rule L^{τ}-valid trace and by Rule L^{τ}-Monitor Step we need to show that $\vdash \mathsf{H} : \Delta$.

This follows by HPH.

We thus need to prove that the initial steps are related heaps are secure.

By Rule L^{τ}-Plug we need to show that the heaps consituting the initial heap – both H and H_0 – are well typed.

The latter, $\vdash \mathsf{H}_0 : \Delta$, holds by Rule L^{τ}-Plug.

The former holds by definition of the attacker: Rules TUL^{τ}-base and TUL^{τ}-loc. □

♠

12.6.1 Proof of Lemma 4 (Semantics and typed attackers coincide)

Proof. This is proven by trivial induction on the syntax of A.

By the rules of Section 5.1.3, points 1 and 3 follow, point 2 follows from the HP Rule L^{τ}-Plug. □

♠

Lemma 12 (L^{τ}-α reductions respect heap typing)**.**

$$\begin{aligned}
\text{if } & \mathsf{C} \equiv \Delta; \cdots \\
& \vdash \texttt{mon-care}(\mathsf{H}, \Delta) : \Delta \\
& \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xrightarrow{\overline{\alpha}} \mathsf{C}', \mathsf{H}' \triangleright \Pi'\rho' \\
\text{then } & \vdash \texttt{mon-care}(\mathsf{H}', \Delta) : \Delta
\end{aligned}$$

Proof. The proof proceeds by induction on $\overline{\alpha}$.

Base case This trivially holds by HP.

Inductive case This holds by IH plus a case analysis on the last action:

$\texttt{call f v?}$ This holds by Lemma 17 (L^{τ}-? actions respect heap typing).

$\texttt{call f v!}$ This holds by Lemma 18 (L^{τ}-! actions respect heap typing)

$\texttt{ret !}$ This holds by Lemma 18 (L^{τ}-! actions respect heap typing)

$\texttt{ret ?}$ This holds by Lemma 17 (L^{τ}-? actions respect heap typing)

□

♠

Lemma 13 (L^{τ} An attacker only reaches UN locations)**.**

$$\begin{aligned}
& \forall \\
\text{if } & \ell \mapsto \mathsf{v} : \mathsf{UN} \in \mathsf{H} \\
\text{then } & \nexists \mathsf{e} \\
& \mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \ell' \\
& \ell' \mapsto \mathsf{v} : \tau \in \mathsf{H} \\
& \tau \neq \mathsf{UN}
\end{aligned}$$

Proof. This proof proceeds by contradiction.
Suppose e exists, there are two cases for ℓ'

- ℓ' was allocated by the attacker:

  This contradicts the judgements of Section 5.1.3.

- ℓ' was allocated by the compiled code:

  The only way this was possible was an assignment of ℓ' to ℓ, but Rule TL^{τ}-assign prevents it.

□

♠

Lemma 14 (L^τ attacker reduction respects heap typing)**.**

$$\begin{aligned} \text{if } & \mathsf{C} \equiv \Delta; \cdots \\ & \mathsf{C} \vdash_{\mathsf{att}} \Pi \longrightarrow \Pi' \\ & \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xrightarrow{\lambda} \mathsf{C}, \mathsf{H}' \triangleright \Pi\rho' \\ \text{then } & \texttt{mon-care}(\mathsf{H}, \Delta) = \texttt{mon-care}(\mathsf{H}', \Delta) \end{aligned}$$

Proof. Trivial induction on the derivation of Π, which is typed with $\vdash_{\mathsf{UN}}$ and by Lemma 13 (L^τ An attacker only reaches UN locations) has no access to locations in Δ or with a type $\tau \vdash \circ$. □

♠

Lemma 15 (L^τ typed reduction respects heap typing)**.**

$$\begin{aligned} \text{if } & \mathsf{C} \equiv \Delta; \cdots \\ & \mathsf{C}, \Gamma \vdash \mathsf{s} \\ & \mathsf{C}, \Gamma \vdash \mathsf{s}' \\ & \vdash \texttt{mon-care}(\mathsf{H}, \Delta) : \Delta \\ & \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}\rho \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \triangleright \mathsf{s}'\rho' \\ \text{then } & \vdash \texttt{mon-care}(\mathsf{H}', \Delta) : \Delta \end{aligned}$$

Proof. This is done by induction on the derivation of the reducing statement.

There, the only non-trivial cases are:

Rule TL^τ-new By IH we have that

$\mathsf{H} \triangleright \mathsf{e}\rho \lhook\joinrel\twoheadrightarrow \mathsf{v}$

So

$\mathsf{C}; \mathsf{H} \triangleright \mathsf{let}\ \mathsf{x} = \mathsf{new}_\tau\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H}\ell \mapsto \mathsf{v} : \tau \triangleright \mathsf{s}[\ell \mathbin{/} \mathsf{x}]$

By IH we need to prove that $\vdash \texttt{mon-care}(\ell \mapsto \mathsf{v} : \tau, \Delta) : \Delta$

As $\ell \notin \texttt{dom}(\Delta)$, by Rule L^τ-Heap-ok-i this case holds.

Rule TL^τ-assign By IH we have (HPH) $\mathsf{H} \triangleright \mathsf{e} \lhook\joinrel\twoheadrightarrow \mathsf{v}$

such that $\ell : \mathsf{Ref}\ \tau$ and $\mathsf{v} : \tau$.

So

$\mathsf{C}; \mathsf{H} \triangleright \mathsf{x} := \mathsf{e}\rho \xrightarrow{\epsilon} \mathsf{C}; \mathsf{H}' \triangleright \mathsf{skip}$

where $[\mathsf{x} \mathbin{/} \ell] \in \rho$ and

$\mathsf{H} = \mathsf{H}_1; \ell \mapsto \mathsf{v}' : \tau; \mathsf{H}_2$

$\mathsf{H}' = \mathsf{H}_1; \ell \mapsto \mathsf{v} : \tau; \mathsf{H}_2$

There are two cases

$\ell \in \texttt{dom}(\Delta)$ By Rule L^{τ}-Heap-ok-i we need to prove that $\ell : \mathsf{Ref}\ \tau \in \Delta$.

This holds by HPH and Rule L^{τ}-Initial State, as the initial state ensures that location ℓ in the heap has the same type as in Δ .

$\ell \notin \texttt{dom}(\Delta)$ This case is trivial as for allocation.

Rule TL^{τ}-coercion We have that $\mathsf{C}, \Gamma \vdash \mathsf{e} : \tau$ and HPT $\tau \vdash \circ$.

By IH $\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\rightarrow \mathsf{v}$ such that $\vdash \texttt{mon-care}(\mathsf{H}', \Delta) : \Delta$.

By HPT we get that $\texttt{mon-care}(\mathsf{H}) = \texttt{mon-care}(\mathsf{H}')$ as by Rule L^{τ}-Secure heap function $\texttt{mon-care}(\cdot)$ only considers locations whose type is $\tau \nvdash \circ$, so none affected by e.

So this case by IH.

Rule TL^{τ}-endorse By Rule EL^{τ}-endorse we have that $\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\rightarrow \mathsf{v}$ and that $\mathsf{C}, \mathsf{H} \triangleright \mathsf{endorse}\ \mathsf{x} = \mathsf{e}\ \mathsf{as}\ \varphi\ \mathsf{in}\ \mathsf{s} \hookrightarrow\!\!\!\rightarrow \mathsf{C}, \mathsf{H} \triangleright \mathsf{s}[\mathsf{v} \mathbin{/} \mathsf{x}]$.

So this holds by IH.

□

♠

Lemma 16 (L^{τ} any non-cross reduction respects heap typing)**.**

$$\begin{aligned}
\text{if } & \mathsf{C} \equiv \Delta; \cdots \\
& \vdash \texttt{mon-care}(\mathsf{H}, \Delta) : \Delta \\
& \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \triangleright \Pi'\rho' \\
\text{then } & \vdash \texttt{mon-care}(\mathsf{H}', \Delta) : \Delta
\end{aligned}$$

Proof. By induction on the reductions and by application of Rule EL^{τ}-par. The base case follows by the assumptions directly. In the inductive case we have the following:

$$\mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xrightarrow{\lambda} \mathsf{C}'', \mathsf{H}'' \triangleright \Pi''\rho'' \xrightarrow{\lambda} \mathsf{C}', \mathsf{H}' \triangleright \Pi'\rho'$$

This has 2 sub-cases, if the reduction is in an attacker function or not.

$\mathsf{C} \vdash_{\mathsf{att}} \Pi'' \longrightarrow \Pi$**:** this follows by induction on Π'' and from IH and Lemma 14 (L^{τ} attacker reduction respects heap typing).

$\mathsf{C} \nvdash_{\mathsf{att}} \Pi'' \longrightarrow \Pi$**:** In this case we induce on Π''.

The base case is trivial.

The inductive case is $(\mathsf{s})_{\bar{\mathsf{f}}} \parallel \Pi$, which follows from IH and Lemma 15 (L^{τ} typed reduction respects heap typing).

□

Lemma 17 (L^{τ}-? actions respect heap typing)**.**

$$\begin{aligned}&\text{if } \mathsf{C} \equiv \Delta; \cdots\\ &\quad \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xRightarrow{\alpha?} \mathsf{C}, \mathsf{H}' \triangleright \mathsf{v}'\\ &\text{then } \texttt{mon-care}(\mathsf{H}, \Delta) = \texttt{mon-care}(\mathsf{H}', \Delta)\end{aligned}$$

Proof. By Lemma 16 (L^{τ} any non-cross reduction respects heap typing), and a simple case analysis on $\alpha?$ (which does not modify the heap). □

Lemma 18 (L^{τ}-! actions respect heap typing)**.**

$$\begin{aligned}&\text{if } \mathsf{C} \equiv \Delta; \cdots\\ &\quad \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xRightarrow{\alpha!} \mathsf{C}', \mathsf{H}' \triangleright \mathsf{v}'\\ &\quad \vdash \texttt{mon-care}(\mathsf{H}, \Delta) : \Delta\\ &\text{then } \vdash \texttt{mon-care}(\mathsf{H}', \Delta) : \Delta\end{aligned}$$

Proof. By Lemma 16 (L^{τ} any non-cross reduction respects heap typing) and a simple case analysis on $\alpha!$ (which does not modify the heap). □

12.7 Proof of Theorem 6 (Compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is *CC*)

Proof. By definition initial states have related components, related heaps and well-typed, related starting processes, for $\beta_0 = (\texttt{dom}(\Delta), \texttt{dom}(\mathbf{H_0}), \mathbf{H_0}.\eta)$ so we have:

HRS $\Omega_0\,(\mathsf{C}) \approx_{\beta_0} \mathbf{\Omega_0}\left([\![\mathsf{C}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\right)$.

As the languages have no notion of internal nondeterminism we can apply Lemma 20 (Generalised compiler correctness for $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$) with HRS to conclude. □

Lemma 19 (Expressions compiled with $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$ are related)**.**

$$\begin{aligned}
&\forall\\
\text{if }\ & \mathsf{H} \approx_\beta \mathbf{H}\\
& \mathsf{H} \triangleright \mathsf{e}\rho \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}\\
\text{then }\ & \mathbf{H} \triangleright [\![\mathsf{e}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}[\![\rho]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\mathsf{v}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}
\end{aligned}$$

Proof. The proof is analogous to that of Lemma 7 (Expressions compiled with $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ are related) as the compilers perform the same steps and expression reductions are atomic. □

♠

Lemma 20 (Generalised compiler correctness for $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$)**.**

$$\begin{aligned}
&\forall ... \exists \beta'\\
\text{if }\ & \mathsf{C}; \Gamma \vdash \Pi,\\
& \vdash \mathsf{C} : \mathsf{whole}\\
& \mathsf{C} = \Delta; \overline{\mathsf{F}}; \overline{\mathsf{I}}\\
& [\![\mathsf{C}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} = \mathbf{H_0}; \overline{\mathbf{F}}; \overline{\mathbf{I}} = \mathbf{C}\\
& \mathsf{C}, \mathsf{H} \triangleright \Pi \approx_\beta \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
& \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \Longrightarrow \mathsf{C}, \mathsf{H}' \triangleright \Pi'\rho'\\
\text{then }\ & \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}[\![\rho]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \Longrightarrow \mathbf{C}, \mathbf{H}' \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi']\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}[\![\rho']\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
& \mathsf{C}, \mathsf{H} \triangleright \Pi'\rho' \approx_{\beta'} \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi']\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}[\![\rho']\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
& \beta \subseteq \beta'
\end{aligned}$$

Proof. This proof proceeds by induction on the typing of Π and then of π.

Base Case skip Trivial by Rule ($[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$-Skip).

Inductive Case

In this case we proceed by induction on the typing of s

Inductive Cases Rule TL^τ-new There are 2 cases, they are analogous.

$\tau = \mathsf{UN}$ By HP

$\Gamma \vdash \mathsf{e} : \tau$

$\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}$

$\mathsf{C}, \mathsf{H} \triangleright \mathsf{let}\ \mathsf{x} = \mathsf{new}_\tau\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}\rho \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H}; \ell \mapsto \mathsf{v} : \tau \triangleright \mathsf{s}[\ell \,/\, \mathsf{x}]\rho$

By Lemma 19 we have:

IHR1 $\mathbf{H} \triangleright [\![\Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}[\![\rho]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\Gamma \vdash \mathsf{v} : \tau]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$

By Rule ($\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$-New) we get

$\mathbf{let\ xo\ = new}\ \llbracket\Gamma\vdash \mathsf{e}:\tau\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$
$\mathbf{in\ let\ x} = \langle\mathbf{xo},\mathbf{0}\rangle$
$\mathbf{in}\ \llbracket\mathsf{C},\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$

So:

$$\begin{aligned}
&\mathbf{C},\mathbf{H}\triangleright\mathbf{let\ xo\ = new}\ \llbracket\Gamma\vdash \mathsf{e}:\tau\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
&\qquad\mathbf{in\ let\ x} = \langle\mathbf{xo},\mathbf{0}\rangle\\
&\qquad\mathbf{in}\ \llbracket\mathsf{C},\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
\xrightarrow{\epsilon}\ &\mathbf{C},\mathbf{H};\mathbf{n}\mapsto\llbracket\Gamma\vdash\mathsf{v}:\tau\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}:\bot\triangleright\mathbf{let\ x}=\langle\mathbf{n},\mathbf{0}\rangle\\
&\qquad\mathbf{in}\ \llbracket\mathsf{C},\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
\xrightarrow{\epsilon}\ &\mathbf{C},\mathbf{H};\mathbf{n}\mapsto\llbracket\Gamma\vdash\mathsf{v}:\tau\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}:\bot\triangleright\llbracket\mathsf{C},\Gamma;\mathsf{x}:\mathsf{Ref}\ \tau\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}[\langle\mathbf{n},\mathbf{0}\rangle\ /\ \mathbf{x}]
\end{aligned}$$

For $\beta' = \beta\cup(\ell,\mathbf{n},\bot)$, this case holds.

else The other case holds follows the same reasoning but for $\beta' = \beta\cup(\ell,\mathbf{n},\mathbf{k})$ and for $\mathbf{H'}{=}\mathbf{H};\mathbf{n}\mapsto\llbracket\mathsf{C},\Gamma\vdash\mathsf{v}:\tau\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}:\mathbf{k};\mathbf{k}$.

Rule TL^τ-sequence By HP

$\Gamma\vdash\mathsf{s};\ \Gamma\vdash\mathsf{s}'$
$\mathsf{C},\mathsf{H}\triangleright\mathsf{s}\rho\Longrightarrow\mathsf{C}',\mathsf{H}'\triangleright\mathsf{s}''\rho''$

There are two cases

$\mathsf{s}''{=}\mathsf{skip}$ Rule EL^U-sequence

$\mathsf{C}',\mathsf{H}'\triangleright\mathsf{skip}\rho'';\mathsf{s}'\rho\xrightarrow{\epsilon}\mathsf{C}',\mathsf{H}'\triangleright\mathsf{s}'\rho$

By IH

$\mathbf{C},\mathbf{H}\triangleright\llbracket\Gamma\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\Longrightarrow\mathbf{C}',\mathbf{H}'\triangleright\llbracket\Gamma\vdash\mathsf{skip}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho''\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$

By Rule ($\llbracket\cdot\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$-Seq)

$\llbracket\mathsf{C},\Gamma\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi};\llbracket\mathsf{C},\Gamma\vdash\mathsf{s}'\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$

So

$$\begin{aligned}
&\mathbf{C},\mathbf{H}\triangleright\llbracket\mathsf{C},\Gamma\vdash\mathsf{s}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi};\llbracket\mathsf{C},\Gamma\vdash\mathsf{s}'\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
\Longrightarrow\ &\mathbf{C}',\mathbf{H}'\triangleright\llbracket\Gamma\vdash\mathsf{skip}\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho''\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi};\llbracket\mathsf{C},\Gamma\vdash\mathsf{s}'\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\\
\xrightarrow{\epsilon}\ &\mathbf{C}',\mathbf{H}'\triangleright\llbracket\mathsf{C},\Gamma\vdash\mathsf{s}'\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\llbracket\rho\rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}
\end{aligned}$$

At this stage we apply IH and the case holds.

else By Rule EL^U-step we have

$\mathsf{C},\mathsf{H}\triangleright\mathsf{s};\mathsf{s}'\Longrightarrow\mathsf{C}',\mathsf{H}'\triangleright\mathsf{s}'';\mathsf{s}'$

This case follows by IH and HPs.

Rule TL^τ-function-call Analogous to the cases above.

Rule TL^τ-letin Analogous to the cases above.

Rule TL^τ-assign Analogous to the cases above.

Rule TL^τ-if Analogous to the cases above.

Rule $\mathbf{T}\mathsf{L}^{\tau}$-fork Analogous to the cases above.

Rule $\mathbf{T}\mathsf{L}^{\tau}$-coercion By Rule ($[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-Coerce), this follows from IH directly.

Rule $\mathbf{T}\mathsf{L}^{\tau}$-endorse This has a number of trivial cases based on Rule ($[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-Endorse) that are analogous to the ones above.

□

♠

12.8 Proof of Theorem 7 (Compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ is *RSC*)

Proof. Given:

HP1: $\mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$

HPM: $\mathsf{M} \approx_{\beta} \mathbf{M}$

We need to prove:

TP1: $\mathbf{M} \vdash [\![\mathsf{C}]\!]^{\mathsf{S}}_{\mathbf{T}} : \mathbf{rs}$

We unfold the definitions of rs and obtain:

$\forall \mathsf{A}.\mathsf{M} \vdash \mathsf{A} : \mathsf{attacker}, \vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}$

HPE1: if $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \_$ then $\mathsf{M} \vdash \mathtt{heaps}(\overline{\alpha})$

$\forall \mathbf{A}.\mathbf{M} \vdash \mathbf{A} : \mathbf{attacker}, \vdash \mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\right] : \mathbf{whole}$

THE1: if HPRT $\mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\right]\right) \xRightarrow{\overline{\alpha}} \_$ then THE1 $\mathbf{M} \vdash \mathtt{heaps}(\overline{\alpha})$

By definition of the compiler we have that

HPISR: $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \approxeq_{\beta} \mathbf{\Omega_0}\left(\mathbf{A}\left[[\![\mathsf{C}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\right]\right)$

for $\beta = \mathtt{dom}(\Delta)$, $\mathbf{H_0}$ such that $\mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_c)$ and $\mathbf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_{\mathbf{0}}, \mathbf{H_0}, \sigma_{\mathbf{c}})$

By $\mathtt{heaps}(\overline{\alpha})$ and Rule $\mathbf{L}^{\pi}$-valid trace we get a $\overline{\mathbf{H}}$ to induce on.

Base case: this holds by Rule $\mathbf{L}^{\pi}$-Monitor Step Trace Base.

Inductive case: By Rule $\mathbf{L}^{\pi}$-Monitor Step Trace, $\mathbf{M}; \overline{\mathbf{H}} \rightsquigarrow \mathbf{M}''$ holds by IH, we need to prove $\mathbf{M}''; \mathbf{H} \rightsquigarrow \mathbf{M}'$.

By Rule $\mathbf{L}^{\pi}$-Monitor Step e need to prove that THMR: $\exists \sigma'.(\sigma, \mathtt{mon\text{-}care}(\mathbf{H}, \mathbf{H_0}), \sigma') \in \rightsquigarrow$.

By HPISR and with applications of Lemmas 22 and 23 we know that states are always related with $\approxeq_{\beta}$ during reduction.

So by Lemma 21 ($\approxeq_{\beta}$ implies relatedness of the high heaps) we know that HPHH $\mathtt{mon\text{-}care}(\mathsf{H}, \Delta) \approx_{\beta} \mathtt{mon\text{-}care}(\mathbf{H}, \mathbf{H_0})$, for H, $\mathbf{H}$ being the last heaps in the reduction.

By HPM and Rule Monitor relation we have $\beta_0, \Delta \vdash \mathbf{M}$.

By this and Rule Ok Mon we have that HPHR $\forall \mathtt{mon\text{-}care}(\mathsf{H}, \Delta) \approx_{\beta} \mathtt{mon\text{-}care}(\mathbf{H}, \mathbf{H_0})$. if $\vdash \mathsf{H} : \Delta$ then $\exists \sigma'.(\sigma, \mathtt{mon\text{-}care}(\mathbf{H}, \mathbf{H_0}), \sigma') \in \rightsquigarrow$ so by HPHH we can instantiate this with H and $\mathbf{H}$.

By Theorem 5 (Typability Implies Robust Safety in L^{τ}) applied to HPE1, as $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ operates on well-typed components, we know that HPMR: $\mathsf{M} \vdash \mathtt{heaps}(\overline{\alpha})$ for all $\overline{\alpha}$.

So by Rule L^{τ}-Monitor Step with HPMT we get HPHD $\vdash \mathsf{H} : \Delta$ for the H above.

By HPHD with HPHR we get THMR $\exists \sigma'.(\sigma, \mathtt{mon\text{-}care}(\mathbf{H}, \mathbf{H_0}), \sigma') \in \rightsquigarrow$, so this case holds.

□

♠

Lemma 21 ($\approx_{\beta}$ implies relatedness of the high heaps)**.**

$$\begin{aligned}
\text{if }\ & \Omega = \Delta; \overline{\mathsf{F}}, \overline{\mathsf{F}'}; \overline{\mathsf{I}}; \mathsf{H} \triangleright \Pi \\
& \mathbf{\Omega} = \mathbf{H_0}; \overline{\mathbf{F}}, [\![\overline{\mathsf{F}'}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}; \overline{\mathbf{I}}; \mathbf{H} \triangleright \mathbf{\Pi} \\
& \Omega \approx_{\beta} \mathbf{\Omega} \\
\text{then }\ & \mathtt{mon\text{-}care}(\mathsf{H}, \Delta) \approx_{\beta} \mathtt{mon\text{-}care}(\mathbf{H}, \mathbf{H_0})
\end{aligned}$$

Proof. By point 2a in Rule Related states – Secure. □

♠

Lemma 22 (L^{τ}-compiled actions preserve $\approx_{\beta}$)**.**

$$\begin{aligned}
& \forall \ldots \\
\text{if }\ & \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xrightarrow{\lambda} \mathsf{C}, \mathsf{H}' \triangleright \Pi'\rho' \\
& \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} [\![\rho]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \overset{\lambda}{\Longrightarrow} \mathbf{C}, \mathbf{H}' \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi']\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} [\![\rho']\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \\
& \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \approx_{\beta} \mathbf{C}, \mathbf{H} \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \rho \\
& \mathsf{C}; \Gamma \vdash \Pi \\
\text{then }\ & \mathsf{C}, \mathsf{H}' \triangleright \Pi'\rho' \approx_{\beta} \mathbf{C}, \mathbf{H}' \triangleright [\![\mathsf{C}; \Gamma \vdash \Pi']\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} [\![\rho']\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}
\end{aligned}$$

Proof. Trivial induction on the derivation of Π, analogous to Lemma 20 (Generalised compiler correctness for $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$).

Rule TL^{τ}-new There are 2 cases, they are analogous.

$\tau = \mathsf{UN}$ By HP

$\Gamma \vdash \mathsf{e} : \tau$

$\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\twoheadrightarrow \mathsf{v}$

$\mathsf{C}, \mathsf{H} \triangleright \mathsf{let}\ \mathsf{x} = \mathsf{new}_{\tau}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}\rho \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H}; \ell \mapsto \mathsf{v} : \tau \triangleright \mathsf{s}[\ell\ /\ \mathsf{x}]\rho$

By Lemma 19 (Expressions compiled with $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ are related) we have:

IHR1 $\mathbf{H} \triangleright [\![\Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} [\![\rho]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \hookrightarrow\!\!\!\!\twoheadrightarrow [\![\Gamma \vdash \mathsf{v} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$

By Rule ($[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$-New) we get

$$\begin{aligned}&\mathbf{let\ xo\ = new}\ [\![\Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ &\quad \mathbf{in\ let\ x} = \langle \mathbf{xo}, \mathbf{0} \rangle\\ &\quad \mathbf{in}\ [\![\mathsf{C}, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\end{aligned}$$

So:

$$\begin{aligned}&\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ xo\ = new}\ [\![\Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ &\qquad\qquad \mathbf{in\ let\ x} = \langle \mathbf{xo}, \mathbf{0} \rangle\\ &\qquad\qquad \mathbf{in}\ [\![\mathsf{C}, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \xrightarrow{\epsilon}\ &\mathbf{C}, \mathbf{H}; \mathbf{n} \mapsto [\![\Gamma \vdash \mathsf{v} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : \bot \triangleright \mathbf{let\ x} = \langle \mathbf{n}, \mathbf{0} \rangle\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ [\![\mathsf{C}, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\ \xrightarrow{\epsilon}\ &\mathbf{C}, \mathbf{H}; \mathbf{n} \mapsto [\![\Gamma \vdash \mathsf{v} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : \bot \triangleright [\![\mathsf{C}, \Gamma; \mathsf{x} : \mathsf{Ref}\ \tau \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}[\langle \mathbf{n}, \mathbf{0} \rangle\ /\ \mathbf{x}]\end{aligned}$$

For $\beta' = \beta$, this case holds.

else The other case holds follows the same reasoning but for $\beta' = \beta \cup (\ell, \mathbf{n}, \mathbf{k})$ and for $\mathbf{H'}{=}\mathbf{H}; \mathbf{n} \mapsto [\![\mathsf{C}, \Gamma \vdash \mathsf{v} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} : \mathbf{k}; \mathbf{k}$.
We need to show that this preserves Rule Related states – Secure, specifically it preserves point $(2a)$: $\ell \approx_{\beta} \langle \mathbf{n}, \mathbf{k} \rangle$ and $\ell \mapsto \mathsf{v} : \tau \in \mathsf{H}$ and $\mathsf{v} \approx_{\beta} \mathbf{v}$
These follow all from the observation above and by Lemma 19 (Expressions compiled with $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ are related).

□

♠

Lemma 23 ($\mathbf{L^P}$ Attacker actions preserve $\approxeq$)**.**

$$\begin{aligned}&\forall \ldots\\ &\text{if}\ \ \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \xrightarrow{\lambda} \mathsf{C}, \mathsf{H}' \triangleright \Pi'\rho'\\ &\quad\ \ \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi}\rho \xrightarrow{\lambda} \mathbf{C}, \mathbf{H}' \triangleright \mathbf{\Pi}'\rho'\\ &\quad\ \ \mathsf{C}, \mathsf{H} \triangleright \Pi\rho \approxeq_{\beta} \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi}\rho\\ &\quad\ \ \mathsf{C} \vdash_{\mathsf{att}} \Pi\rho \xrightarrow{\lambda} \Pi'\rho'\end{aligned}$$

$$\mathbf{C} \vdash_{\mathbf{att}} \mathbf{\Pi}\rho \xrightarrow{\lambda} \mathbf{\Pi}'\rho'$$
$$\text{then} \quad \mathsf{C}, \mathsf{H}' \triangleright \mathsf{\Pi}'\rho' \approx_\beta \mathbf{C}, \mathbf{H}' \triangleright \mathbf{\Pi}'\rho'$$

Proof. For the source reductions we can use Lemma 16 (L^τ any non-cross reduction respects heap typing) to know that $\texttt{mon-care}(\mathsf{H}) = \texttt{mon-care}(\mathsf{H}')$, so they don't change the interested bits of the $\approx_\beta$.

Suppose this does not hold by contradiction, there can be three clauses that do not hold based on Rule Related states – Secure:

- violation of (1): $\exists \pi \in \mathbf{\Pi}. \mathsf{C} \vdash \pi : \mathbf{attacker}$ and $\mathbf{k} \in \texttt{fv}(\pi)$.

  By HP5 this is a contradiction.

- violation of $(2a)$: $\mathbf{n} \mapsto \mathbf{v} : \mathbf{k} \in \mathbf{H}$ and $\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle$ and $\ell \mapsto \mathsf{v} : \tau \in \mathsf{H}$ and $\neg(\mathsf{v} \approx_\beta \mathbf{v})$

  To change this value the attacker needs $\mathbf{k}$ which contradicts points (1) and $(2b)$.

- violation of $(2b)$: either of these:

  - $\mathsf{H}, \mathbf{H} \nvdash \texttt{low-loc}(\mathbf{n}')$

    Since Rule $\mathbf{L}^\pi$-High Location does not hold, by Lemma 5 this is a contradiction.

  - $\mathbf{v} = \mathbf{k}'$ for $\mathsf{H}, \mathbf{H} \vdash \texttt{high-cap}(\mathbf{k}')$

    This can follow from another two cases

    * forgery of $\mathbf{k}$;: an ispection of the semantics rules contradicts this
    * update of a location to $\mathbf{k}'$: however $\mathbf{k}'$ is not in the code (contradicts point (1)) and by induction on the heap $\mathbf{H}$ we have that $\mathbf{k}'$ is stored in no other location, so this is also a contradiction.

□

♠

12.9 Proofs for the Non-Atomic Variant of L^τ (Section 8.2)

The only proof that needs changing is that for Lemma 22: there is this new case.

For this we weaken $\approx_\beta$ and define $\sim_\beta$ as follows:

$$\boxed{\Omega \sim_\beta \mathbf{\Omega}}$$

$$\text{(Non Atomic State Relation)}\quad \frac{\Omega \approx_\beta \mathbf{\Omega}}{\Omega \sim_\beta \mathbf{\Omega}}$$

$$\text{(Non Atomic State Relation -stuck)}\quad \frac{\begin{array}{c}\Omega = \mathsf{C}, \mathsf{H} \triangleright \Pi \qquad \mathsf{C} = \Delta, \overline{\mathsf{F}}, \overline{\mathsf{I}} \qquad \mathbf{\Omega} = \mathbf{C}, \mathbf{H} \triangleright \mathbf{\Pi} \\ \exists \pi \in \mathbf{\Pi}.\, \mathbf{C} \nvdash \pi : \mathbf{attacker} \\ \pi = (\mathbf{hide\ n; s})_{\overline{\mathbf{f}};\mathbf{f}} \qquad \mathbf{C}, \mathbf{H} \triangleright \pi^{\times} \qquad \exists \mathsf{f} \in \mathtt{dom}(\overline{\mathsf{F}}).\, \mathsf{f} \approx_\beta \mathbf{f} \\ \forall \ell.\, \ell \in \mathtt{dom}(\vdash \mathtt{secure}(\mathsf{H})) \qquad \mathbf{n} \mapsto \mathbf{v}; \mathbf{k} \in \mathbf{H} \qquad \ell \not\sim_\beta \langle \mathbf{n}, \mathbf{k} \rangle \qquad \ell \sim_\beta \langle \mathbf{n}, \mathbf{0} \rangle\end{array}}{\Omega \sim_\beta \mathbf{\Omega}}$$

Two states are now related if:

- either they are related by $\approx_\beta$
- or the red process is stuck on a $\mathbf{hide\ n}$ where $\mathbf{n} \mapsto \mathbf{v}; \mathbf{k}$ but $\ell \sim \langle \mathbf{n}, \mathbf{k} \rangle$ does not hold for a ℓ that is secure, and we have that $\ell \sim \langle \mathbf{n}, \mathbf{0} \rangle$ (as this was after the $\mathbf{new}$). And the $\mathbf{hide}$ on which the process is stuck is not in attacker code.

Having this in proofs would not cause problems because now all proofs have an initial case analysis whether the state is stuck or not, but because it steps it's not stuck.

This relation only changes the second case of the proof of Lemma 22 for Rule ($\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$-New-nonat) as follows:

Proof. $\mathsf{new.}\ \cdot$ is implemented as defined in Rule ($\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$-New-nonat).

$\tau \neq \mathsf{UN}$ By HP

$\Gamma \vdash \mathsf{e} : \tau$

$\mathsf{H} \triangleright \mathsf{e} \hookrightarrow\!\!\!\!\rightarrow \mathsf{v}$

$\mathsf{C}, \mathsf{H} \triangleright \mathsf{let\ x = new}_\tau\ \mathsf{e\ in\ s}\rho \xrightarrow{\epsilon} \mathsf{C}, \mathsf{H}; \ell \mapsto \mathsf{v} : \tau \triangleright \mathsf{s}[\ell \,/\, \mathsf{x}]\rho$

By Lemma 19 we have:

IHR1 $\mathbf{H} \triangleright \llbracket \Gamma \vdash \mathsf{e} : \tau \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \llbracket \rho \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi} \hookrightarrow\!\!\!\!\rightarrow \llbracket \Gamma \vdash \mathsf{v} : \tau \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$

By Rule ($\llbracket \cdot \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$-New-nonat) we get

$\mathbf{let\ x\ \ = new\ 0\ in}$

$\mathbf{let\ xk\ \ = hide\ x\ in}$

$\mathbf{let\ xc} = \llbracket \Delta, \Gamma \vdash \mathsf{e} : \tau \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\ \mathbf{in}$

$\mathbf{x := xc\ with\ xk;}$

$\llbracket \mathsf{C}, \Delta, \Gamma \vdash \mathsf{s} \rrbracket^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$

So:

$$\begin{aligned}
&\mathbf{C}, \mathbf{H} \triangleright \mathbf{let\ x} \ = \mathbf{new\ 0\ in}\\
&\qquad \mathbf{let\ xk} \ = \mathbf{hide\ x\ in}\\
&\qquad \mathbf{let\ xc} = [\![\Delta, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \ \mathbf{in}\\
&\qquad \mathbf{x := xc\ with\ xk;}\\
&\qquad [\![\mathsf{C}, \Delta, \Gamma \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}\\
\xrightarrow{\epsilon}\ &\mathbf{C}, \mathbf{H}, \mathbf{n} \mapsto \mathbf{0} : \bot \triangleright \mathbf{let\ xk} \ = \mathbf{hide\ n\ in}\\
&\qquad \mathbf{let\ xc} = [\![\Delta, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \ \mathbf{in}\\
&\qquad \mathbf{x := xc\ with\ xk;}\\
&\qquad [\![\mathsf{C}, \Delta, \Gamma \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}
\end{aligned}$$

And $\beta' = \beta \cup (\ell, \mathbf{n}, \mathbf{0})$.

Now there are two cases:

- A concurrent attacker reduction performs **hide n**, so the state changes.

$$\begin{aligned}
&\mathbf{C}, \mathbf{H}, \mathbf{n} \mapsto \mathbf{0} : \mathbf{k}; \mathbf{k} \triangleright \mathbf{let\ xk} \ = \mathbf{hide\ n\ in}\\
&\qquad \mathbf{let\ xc} = [\![\mathsf{C}, \Gamma \vdash \mathsf{e} : \tau]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}} \ \mathbf{in}\\
&\qquad \mathbf{x := xc\ with\ xk;}\\
&\qquad [\![\mathsf{C}, \Delta, \Gamma \vdash \mathsf{s}]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}
\end{aligned}$$

  At this stage the state is stuck: Rule $\mathbf{EL}^{\mathbf{P}}$-hide does not apply.
  Also, we have that this holds by the new β': $(\ell \sim_{\beta'} \langle \mathbf{n}, \mathbf{0} \rangle)$
  And so this does not hold: $(\ell \sim_{\beta'} \langle \mathbf{n}, \mathbf{k} \rangle)$
  As the stuck statement is not in attacker code, we can use Rule Non Atomic State Relation -stuck to conclude.

- The attacker does not. In this case the proof continues as in Lemma 22.

□

♠

12.10 Proof of Theorem 8 (Compiler $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{L^{I}}$ is CC)

Proof. Analogous to that of Section 12.7. □

12.11 Proof of Theorem 9 (Compiler $[\![\cdot]\!]^{\mathsf{L}^\tau}_{L^I}$ is RSC)

Proof. Given:

HP1: $\mathsf{M} \vdash \mathsf{C} : \mathsf{rs}$
HPM: $\mathsf{M} \approx_\varphi M$
We need to prove:
TP1: $M \vdash [\![\mathsf{C}]\!]^{\mathsf{L}^\tau}_{L^I} : rs$
We unfold the definitions of rs and obtain:
$\forall \mathsf{A}.\mathsf{M} \vdash \mathsf{A} : \mathsf{attacker}, \vdash \mathsf{A}\left[\mathsf{C}\right] : \mathsf{whole}$
HPE1: if $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \xRightarrow{\overline{\alpha}} \_$ then $\mathsf{M} \vdash \mathtt{heaps}(\overline{\alpha})$
$\forall A.M \vdash A : \mathit{attacker}, \vdash A\left[[\![\mathsf{C}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\right] : \mathit{whole}$
THE1: if HPRT $\Omega_0\left(A\left[[\![\mathsf{C}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\right]\right) \xRightarrow{\overline{\alpha}} \_$ then THE1 $M \vdash \mathtt{heaps}(\overline{\alpha})$
By definition of the compiler we have that
HPISR: $\Omega_0\left(\mathsf{A}\left[\mathsf{C}\right]\right) \approxeq_\varphi \Omega_0\left(A\left[[\![\mathsf{C}]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}\right]\right)$
for $\varphi = \mathtt{dom}(\Delta)$, H_0 such that $\mathsf{M} = (\{\sigma\}, \rightsquigarrow, \sigma_0, \Delta, \sigma_\mathsf{c})$ and $M = (\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c)$
By $\mathtt{heaps}(\overline{\alpha})$ and Rule L^I-valid trace we get a $\overline{H}$ to induce on.

Base case: this holds by Rule L^I-Monitor Step Trace Base.

Inductive case: By Rule L^I-Monitor Step Trace, $M; \overline{H} \rightsquigarrow M''$ holds by IH, we need to prove $M''; H \rightsquigarrow M'$.

By Rule L^I-Monitor Step e need to prove that THMR: $\exists \sigma'.(\sigma, \mathtt{mon\text{-}care}(H, H_0), \sigma') \in \rightsquigarrow$.

By HPISR and with applications of Lemmas 25 and 26 we know that states are always related with $\approxeq_\varphi$ during reduction.

So by Lemma 24 ($\approxeq_\varphi$ implies relatedness of the high heaps) we know that HPHH $\mathtt{mon\text{-}care}(\mathsf{H}, \Delta) \approx_\varphi \mathtt{mon\text{-}care}(H, H_0)$, for H, H being the last heaps in the reduction.

By HPM and Rule Monitor relation (adjusted for L^I) we have $\varphi_0, \Delta \vdash M$.

By this and Rule Ok Mon (adjusted for L^I) we have that

HPHR $\forall \mathtt{mon\text{-}care}(\mathsf{H}, \Delta) \approx_\varphi \mathtt{mon\text{-}care}(H, H_0)$. if $\vdash \mathsf{H} : \Delta$ then

$\exists \sigma'.(\sigma, \mathtt{mon\text{-}care}(H, H_0), \sigma') \in \rightsquigarrow$ so by HPHH we can instantiate this with H and H.

By Theorem 5 (Typability Implies Robust Safety in L^τ) applied to HPE1, as $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$ operates on well-typed components, we know that HPMR: $\mathsf{M} \vdash \mathtt{heaps}(\overline{\alpha})$ for all $\overline{\alpha}$.

So by Rule L^τ-Monitor Step with HPMT we get HPHD $\vdash \mathsf{H} : \Delta$ for the H above.

By HPHD with HPHR we get THMR $\exists \sigma'.(\sigma, \mathtt{mon\text{-}care}(H, H_0), \sigma') \in \rightsquigarrow$, so this case holds.

□

♠

Lemma 24 (≈$_\varphi$ implies relatedness of the high heaps)**.**

$$\begin{aligned}
&\text{if } \mathsf{\Omega} = \mathsf{\Delta}; \overline{\mathsf{F}}, \overline{\mathsf{F'}}; \overline{\mathsf{I}}; \mathsf{H} \triangleright \mathsf{\Pi}\\
&\quad \mathit{\Omega} = H_0; \overline{F}, [\![\overline{\mathsf{F'}}]\!]^{\mathsf{L}^\tau}_{L^I}; \overline{I}; \overline{E}; H \triangleright \mathit{\Pi}\\
&\quad \mathsf{\Omega} \approxeq_\varphi \mathit{\Omega}\\
&\text{then } \texttt{mon-care}(\mathsf{H}, \mathsf{\Delta}) \approx_\varphi \texttt{mon-care}(H, H_0)
\end{aligned}$$

Proof. By Rule Related states – Secure. □

♠

Lemma 25 (L^τ-compiled actions preserve ≈$_\varphi$)**.**

$$\begin{aligned}
&\forall \ldots\\
&\text{if } \mathsf{C}, \mathsf{H} \triangleright \mathsf{\Pi}\rho \xrightarrow{\lambda} \mathsf{C}, \mathsf{H'} \triangleright \mathsf{\Pi'}\rho'\\
&\quad C, H \triangleright [\![\mathsf{C}; \mathsf{\Gamma} \vdash \mathsf{\Pi}]\!]^{\mathsf{L}^\tau}_{L^I} [\![\rho]\!]^{\mathsf{L}^\tau}_{L^I} \xRightarrow{\lambda} C, H' \triangleright [\![\mathsf{C}; \mathsf{\Gamma} \vdash \mathsf{\Pi'}]\!]^{\mathsf{L}^\tau}_{L^I} [\![\rho']\!]^{\mathsf{L}^\tau}_{L^I}\\
&\quad \mathsf{C}, \mathsf{H} \triangleright \mathsf{\Pi}\rho \approxeq_\varphi C, H \triangleright [\![\mathsf{C}; \mathsf{\Gamma} \vdash \mathsf{\Pi}]\!]^{\mathsf{L}^\tau}_{L^I} \rho\\
&\quad \mathsf{C}; \mathsf{\Gamma} \vdash \mathsf{\Pi}\\
&\text{then } \mathsf{C}, \mathsf{H'} \triangleright \mathsf{\Pi'}\rho' \approxeq_\varphi C, H' \triangleright [\![\mathsf{C}; \mathsf{\Gamma} \vdash \mathsf{\Pi'}]\!]^{\mathsf{L}^\tau}_{L^I} [\![\rho']\!]^{\mathsf{L}^\tau}_{L^I}
\end{aligned}$$

Proof. Trivial induction on the derivation of $\mathsf{\Pi}$, analogous to Lemma 20 (Generalised compiler correctness for $[\![\cdot]\!]^{\mathsf{L}^\tau}_{\mathbf{L}^\pi}$) and Lemma 22 ($\mathsf{L}^\tau$-compiled actions preserve ≈$_\beta$). □

♠

Lemma 26 ($\mathbf{L}^\mathbf{P}$ Attacker actions preserve ≈)**.**

$$\begin{aligned}
&\forall \ldots\\
&\text{if } \mathsf{C}, \mathsf{H} \triangleright \mathsf{\Pi}\rho \xrightarrow{\lambda} \mathsf{C}, \mathsf{H'} \triangleright \mathsf{\Pi'}\rho'\\
&\quad C, H \triangleright \mathit{\Pi}\rho \xrightarrow{\lambda} C, H' \triangleright \mathit{\Pi'}\rho'\\
&\quad \mathsf{C}, \mathsf{H} \triangleright \mathsf{\Pi}\rho \approxeq_\varphi C, H \triangleright \mathit{\Pi}\rho\\
&\quad \mathsf{C} \vdash_{\mathsf{att}} \mathsf{\Pi}\rho \xrightarrow{\lambda} \mathsf{\Pi'}\rho'\\
&\quad C \vdash_{att} \mathit{\Pi}\rho \xrightarrow{\lambda} \mathit{\Pi'}\rho'\\
&\text{then } \mathsf{C}, \mathsf{H'} \triangleright \mathsf{\Pi'}\rho' \approxeq_\varphi C, H' \triangleright \mathit{\Pi'}\rho'
\end{aligned}$$

Proof. For the source reductions we can use Lemma 16 (L^τ any non-cross reduction respects heap typing) to know that $\texttt{mon-care}(\mathsf{H}) = \texttt{mon-care}(\mathsf{H}')$, so they don't change the interested bits of the $\approxeq_\varphi$.

Suppose this does not hold by contradiction, there can be one clause that does not hold based on Rule Related states – Secure:

- two related high-locations ℓ and n point to unrelated values.

  Two cases arise: creation and update of a location to an unrelated value.

  Both cases are impossible because Rule EL^I-assign-iso and Rule EL^I-isolate check $C \vdash f : prog$ and Rule L^I-Whole ensures that the attacker defines different names from the program, so the attacker can never execute them.

□

13 *FAC* and Inefficient Compiled Code

We illustrate various ways in which *FAC* forces inefficiencies in compiled code via a running example. Consider a password manager written in an object-oriented language that is compiled to an assembly-like language. We elide most code details and focus only on the relevant aspects.

```
private db: Database;

public testPwd( user: Char[8], pwd: BitString): Bool{
 if( db.contains( user )){ return db.get( user ).getPassword() == pwd; }
}
...
private class Database{ ... }
```

The source program exports the function `testPwd` to check whether a `user`'s stored password matches a given password `pwd`. The stored password is in a local database, which is represented by a piece of *local state* in the variable `db`. The details of `db` are not important here, but the database is marked private, so it is not directly accessible to the context of this program in the source language.

Example 1 (Extensive checks)**.** A fully-abstract compiler for the program above must generate code that checks that the arguments passed to `testPwd` by the context are of the right type [2, 6, 10, 14, 16]. In fact, the code expects an array of characters of length 8, any other parameter (e.g., an array of objects) cannot be passed in the source, so it must also be prevented to be passed in the target. More precisely, a fully abstract compiler will generate code similar to the following for `testPwd` (we assume that arrays are passed as pointers into the heap).

```
label testpwd
  for i = 0; i< 8; i++  // 8 is the legth of the user field in the previous snippet
    add r0 i
    load the memory word stored at address r0 into r1
    test that r1 is a valid char encoding
  ...
```

Basically, this code dynamically checks that the first argument is a character array. Such a check can be very inefficient. ⊡

The problem here is that *FAC* forces these checks on all arguments, even those that have no security relevance. In contrast, *RSC* does not need these checks. Indeed, neither of our earlier compiler, $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ nor $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$, insert them. Note that any robustly safe source program will have programmer-inserted checks for all parameters that are relevant to the safety property of interest, and these checks will be compiled to the target. For other parameters, the checks are irrelevant, both in the source and the target, so there is no need to insert them.

Example 2 (Component size in memory)**.** Let us now consider two different ways to implement the `Database` class: as a `List` and as a `RedBlackTree`. As the class is `private`, its internal behaviour and representation of the database is invisible to the outside. Let $\mathsf{C}_{\mathsf{list}}$ be the program with the `List` implementation and $\mathsf{C}_{\mathsf{tree}}$ be the program with the `RedBlackTree` implementation; in the source language, these are equivalent.

However, a subtlety arises when considering the assembly-level, compiled counterparts of $\mathsf{C}_{\mathsf{list}}$ and $\mathsf{C}_{\mathsf{tree}}$: the *code* of a `RedBlackTree` implementation consumes more memory than the code of a `List` implementation. Thus, a target-level context can distinguish $\mathsf{C}_{\mathsf{list}}$ from $\mathsf{C}_{\mathsf{tree}}$ by just inspecting the sizes of the code segments. So, in order for $[\![\cdot]\!]^{\mathsf{S}}_{\mathbf{T}}$ to be fully abstract, it must produce code of a fixed size [2, 14]. This wastes memory and makes it impossible to compile some components. An alternative would be to spread the components in an overly-large memory at random places i.e., use address-space layout randomization or ASLR, so that detecting different code sizes has a negligible chance of success [1, 7]. However, ASLR is now known to be broken [3, 8]. $\boxdot$

Again, we see that *FAC* introduces an inefficiency in compiled code (pointless code memory consumption) even though this has no security implication here. In contrast, *RSC* does not require this unless the safety property(ies) of interest care about the size of the code (which is very unlikely in a security context, since security by code obscurity is a strongly discouraged security practice). In particular, the monitors considered in this paper cannot depend on code size.

Example 3 (Wrappers for heap resources)**.** Assume that the `Database` class is implemented as a `List`. Shown below are two implementations of the `newList` method inside `List` which we call $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$. The only difference between $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ is that $\mathsf{C}_{\mathsf{two}}$ allocates two lists internally; one of these (`shadow`) is used for internal purposes only.

```
public newList(): List{

  ell = new List();
  return ell;
}
```

```
public newList(): List{
  shadow = new List();
  ell = new List();
  return ell;
}
```

Again, $\mathsf{C}_{\mathsf{one}}$ and $\mathsf{C}_{\mathsf{two}}$ are equivalent in a source language that does not allow pointer comparison. To attain *FAC* when the target allows pointer comparisons, the pointers returned by `newList` in the two implementations must be the same, but this is very difficult to ensure since the second implementation does more allocations. A simple solution to this problem is to wrap `ell` in a proxy object and return the proxy [2, 12, 14, 16]. Compiled code needs to maintain a lookup table mapping the proxy to the original object. Proxies must have allocation-independent addresses. Proxies work but they are inefficient due to the need to look up the table on every object access.

Another way to attain *FAC* is to weaken the source language, introducing an operation to distinguish object identities in the source [13]. However, this is a widely discouraged practice, as it changes the source language from what it really is and the implication of such a change may be difficult to fathom for programmers and verifiers. $\boxdot$

In this example, *FAC* forces all privately allocated locations to be wrapped in proxies, however *RSC* does not require this. Our target languages $\mathbf{L}^{\mathbf{P}}$ and $\mathbf{L}^{\pi}$ support address comparison (addresses are natural numbers in their heaps) but $[\![\cdot]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$ and $[\![\cdot]\!]^{\mathsf{L}^{\tau}}_{\mathbf{L}^{\pi}}$ just use capabilities to attain security efficiently. On the other hand, for attaining *FAC*, capabilities alone would be insufficient since they do not hide addresses; proxies would still be required (this point is concretely demonstrated in Section 14).

Example 4 (Strict termination vs divergence)**.** Consider a source language that is strictly terminating while a target language that is not. Below is an extension of the password manager to allow database encryption via an externally-defined function. As the database is not directly accessible from external code, the two implementations below $\mathsf{C}_{\mathsf{enc}}$ (which does the encryption) and $\mathsf{C}_{\mathsf{skip}}$ which skips the encryption are equivalent in the source.

```
public encryptDB( func : Database
    →Bitstring) : void {
  func( this.db );
  return;
}
```

```
public encryptDB( func : Database
    →Bitstring) : void {

  return;
}
```

If we compile $\mathsf{C}_{\mathsf{enc}}$ and $\mathsf{C}_{\mathsf{skip}}$ to an assembly language, the compiled counterparts *cannot* be equivalent, since the target-level context can detect which function is compiled by passing a `func` that diverges. Calling the compilation of $\mathsf{C}_{\mathsf{enc}}$ with such a `func` will cause divergence, while calling the compilation of $\mathsf{C}_{\mathsf{skip}}$ will immediately return. ⊡

This case presents a situation where *FAC* is outright *impossible*. The only way to get *FAC* is to make the source language artificially non-terminating, see the work of Devriese *et al.* **(author?)** [5] for more details of this particular problem. On the other hand, *RSC* can be easily attained even in such settings since it is completely independent of termination in the languages (note that program termination and nontermination are both different from the monitor getting stuck on an action, which is what *RSC* cares about). Indeed, if our source languages L^{U} and L^{τ} were restricted to terminating programs only, the same compilers and the same proofs of *RSC* would still work.

Remark It is worth noting that many of the inefficiencies above could be resolved by just replacing contextual equivalence with a different equivalence in the statement of *FAC*. However, it is not known how to do this generally for arbitrary sources of inefficiency and, further, it is unclear what the security consequences of such instantiations of *FAC* would be. On the other hand, *RSC* is *uniform* and it does address all these inefficiencies.

An issue that can normally not be addressed just by tweaking equivalences is side-channel leaks, as they are, by definition, not expressible in the language. Neither *FAC* nor *RSC* deals with side channels, but recent results describe how to account for side channels in secure compilers [4].

14 Towards a Fully Abstract Compiler from $\mathsf{L^U}$ to $\mathbf{L^P}$

This section sketches a fully abstract compiler from $\mathsf{L^U}$ to $\mathbf{L^P}$.

14.1 Language Extensions to $\mathsf{L^U}$ and $\mathbf{L^P}$

This section lists the language extensions required by the compiler. It is not possible to motivate all of them before explaining the details of the compiler, so some of the justification is postponed to Section 14.2.

A first concern for full abstraction is that a target context can always determine the memory consumption of two compiled components, analogously to Example 2. To ensure that this does not break full abstraction, we add a source expression size that returns the amount of locations ℓ allocated in the current heap H.

In the target language $\mathbf{L^P}$, we need to know whether an expression is a pair, whether it is a location, and we need to be able to compare two capabilities. For this, we add the expression constructs $\mathbf{isloc(e)}$, $\mathbf{ispair(e)}$ and $\mathbf{eqcap(e,e)}$, respectively.

Finally, compiled code needs private functions for its runtime checks that must not be visible to the context. $\mathbf{L^P}$ does not have this functionality: all functions defined by a component can be called by the context. Now we modify $\mathbf{L^P}$ so that all functions $\overline{\mathbf{F}}$ defined in a component are by default private to it. Additionally, each component must explicitly define the list of functions it exports (typically a subset of $\overline{\mathbf{F}}$), so that those are the only ones that can be called by the context and the rest are private to the component.

14.2 The $\wr\cdot\int^{\mathsf{L^U}}_{\mathbf{L^P}}$ Compiler

$\wr\cdot\int^{\mathsf{L^U}}_{\mathbf{L^P}}$ is similar to $\llbracket\cdot\rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$ but with critical differences. We know that fully abstract compilation preserves *all* source abstractions in the target language. Here, the only abstraction that distinguishes $\mathbf{L^P}$ from $\mathsf{L^U}$ is that locations are abstract in $\mathbf{L^P}$, but concrete natural numbers in $\mathsf{L^U}$. Thus, locations allocated by compiled code must not be passed directly to the context as this would reveal the allocation order (as seen in Example 3). Instead of passing the location $\langle\mathbf{n},\mathbf{k}\rangle$ to the context, the compiler arranges for an opaque handle $\langle\mathbf{n'},\mathbf{k_{com}}\rangle$ (that cannot be used to access any location directly) to be passed. Such an opaque handle is often called a *mask* or *seal* in the literature.

To ensure that masking is done properly, $\wr\cdot\int^{\mathsf{L^U}}_{\mathbf{L^P}}$ inserts code at entry points and at exit points to compiled code, *wrapping* the compiled code in a way that enforces masking. This notion of wrapping is standard in literature on fully abstract compilation [6, 16]. The wrapper keeps a list $\overline{\mathbf{L}}$ of component-allocated locations that are shared with the context in order to know their masks. When a component-allocated location is shared, it is added to the list $\overline{\mathbf{L}}$. The mask of a location is its index in this list. If the same location is shared again it is not added again but its previous index is used. So if $\langle \mathbf{n}, \mathbf{k} \rangle$ is the 4th element of $\overline{\mathbf{L}}$, its mask is $\langle 4, \mathbf{k_{com}} \rangle$. To implement lookup in $\overline{\mathbf{L}}$ we must compare capabilities too, so we rely on **eqcap**. To ensure capabilities do not leak to the context, the second field of the pair is a constant capability $\mathbf{k_{com}}$ whose location the compiled code does not use otherwise. Technically speaking, this is exactly how existing fully abstract compilers operate (e.g., [14]).

As should be clear, this kind of masking is very inefficient at runtime. However, even this masking is not sufficient for full abstraction. Next, we explain additional things the compiler must do.

Determining when a Location is Passed to the Context. A component-allocated location can be passed to the context not just as a function argument but on the heap. So before passing control to the context the compiled code needs to scan the whole heap where a location can be passed and mask all found component-allocated locations. Dually, when receiving control the compiled code must scan the heap to unmask it. The problem now is determining what parts of the heap to scan and how. Specifically, the compiled code needs to keep track of all the locations (and related capabilities) that are shared, i.e., (i) passed from the context to the component and (ii) passed from the component to the context. These are the locations on which possible communication of locations can happen. Compiled code keeps track of these shared locations in a list $\overline{\mathbf{S}}$. Intuitively, on the first function call from the context to the compiled component, assuming the parameter is a location, the compiled code will register that location and all other locations reachable from it in $\overline{\mathbf{S}}$. On subsequent ? (incoming) actions, the compiled code will register all new locations available as parameters or reachable from $\overline{\mathbf{S}}$. Then, on any ! (outgoing) action, the compiled code must scan whatever locations (that the compiled code has created) are now reachable from $\overline{\mathbf{S}}$ and add them to $\overline{\mathbf{S}}$. We need the new instructions **isloc** and **ispair** in $\mathbf{L^P}$ to compute these reachable locations. Of course, this kind of scanning of locations reachable from $\overline{\mathbf{S}}$ at every call/return between components can be extremely costly.

Enforcing the Masking of Locations The functions **mask** and **unmask** are added by the compiler to the compiled code. The first function takes a location (which intuitively contains a value $\mathbf{v}$) and replaces (in $\mathbf{v}$) any pair $\langle \mathbf{n}, \mathbf{k} \rangle$ of a location protected with a component-created capability $\mathbf{k}$ with its index in the masking list $\overline{\mathbf{L}}$. The second function replaces any pair $\langle \mathbf{n}, \mathbf{k_{com}} \rangle$ with the nth element of the masking list $\overline{\mathbf{L}}$. These functions should not be directly accessible to the context (else it can **unmask** any **mask**'d location and break full abstraction). This is why $\mathbf{L^P}$ needs private functions.

Letting the Context use Masked Locations Masked locations cannot be used directly by the context to be read and written. Thus, compiled code must provide a **read** and a **write** function (both of which are public) that implement reading and writing to masked locations.

As should be clear, code compiled through $\llparenthesis\cdot\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}}$ has a lot of runtime overhead in calculating the heap reachable from $\overline{\mathbf{S}}$ and in **mask**ing and **unmask**ing locations. Additionally, it also has code memory overhead: the functions **read**, **write**, **mask**, **unmask** and list manipulation code must be included. Finally, there is data overhead in maintaining $\overline{\mathbf{S}}$, $\overline{\mathbf{L}}$ and other supporting data structures to implement the runtime checks described above. In contrast, the code compiled through $[\![\cdot]\!]^{\mathsf{L^U}}_{\mathbf{L^P}}$ (which is just robustly safe and not fully abstract) has none of these overheads.

14.3 Proving that $\llparenthesis\cdot\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}}$ is a Fully Abstract Compiler

Using $\llparenthesis\cdot\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}}$ as a concrete example, we now discuss why *proving FAC* is harder than proving *RSC*. Consider the hard part of *FAC*, the forward implication, $\mathsf{C_1} \simeq_{ctx} \mathsf{C_2} \Rightarrow [\![\mathsf{C_1}]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{C_2}]\!]^{\mathsf{S}}_{\mathbf{T}}$. The contrapositive of this statement is $[\![\mathsf{C_1}]\!]^{\mathsf{S}}_{\mathbf{T}} \not\simeq_{ctx} [\![\mathsf{C_2}]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \mathsf{C_1} \not\simeq_{ctx} \mathsf{C_2}$. By unfolding the definition of $\not\simeq_{ctx}$ we see that, given a target context $\mathbb{C}$ that distinguishes $[\![\mathsf{C_1}]\!]^{\mathsf{S}}_{\mathbf{T}}$ from $[\![\mathsf{C_2}]\!]^{\mathsf{S}}_{\mathbf{T}}$, it is necessary to show that there exists a source context $\mathbb{C}$ that distinguishes $\mathsf{C_1}$ from $\mathsf{C_2}$. That source context $\mathbb{C}$ must be built (backtranslated) starting from the already given target context $\mathbb{C}$ that differentiates $[\![\mathsf{C_1}]\!]^{\mathsf{S}}_{\mathbf{T}}$ from $[\![\mathsf{C_2}]\!]^{\mathsf{S}}_{\mathbf{T}}$.

A backtranslation directed by the syntax of the target context $\mathbb{C}$ is hopeless here since the target expressions **iscap** and **isloc** cannot be directly backtranslated to valid source expressions. Hence, we resort to another well-known technique [2, 16]. First, we define a *fully abstract (labeled) trace semantics* for the target language. A trace semantics is fully abstract when its notion of equivalence coincides with contextual equivalence, and thus can be used in place of the latter. Specifically, this means that two components are contextually inequivalent iff their trace semantics differ in at least one trace. We write $\mathsf{TR}(\mathbf{C})$ to denote the traces of the component $\mathbf{C}$ in this fully abstract semantics. Given this trace semantics, the statement of the forward implication of full abstraction reduces to:

$$\mathsf{TR}(\llparenthesis\mathsf{C_1}\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}}) \neq \mathsf{TR}(\llparenthesis\mathsf{C_2}\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}}) \Rightarrow \mathsf{C_1} \not\simeq_{ctx} \mathsf{C_2}.$$

The advantage of this formulation over the original one is that now we can construct a distinguishing source context for $\mathsf{C_1}$ and $\mathsf{C_2}$ using the *trace* on which $\mathsf{TR}(\llparenthesis\mathsf{C_1}\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}})$ and $\mathsf{TR}(\llparenthesis\mathsf{C_2}\rrparenthesis^{\mathsf{L^U}}_{\mathbf{L^P}})$ disagree. While this proof strategy of constructing a source context from a trace is similar to our proof of *RSC*, it is fundamentally much harder and much more involved. There are two reasons for this.

First, fully abstract trace semantics are much more complex than our simple trace semantics of $\mathbf{L^P}$ from earlier sections. The reason is that our earlier trace semantics include the entire heap in every action, but this breaks full abstraction of the trace semantics: such trace semantics also distinguish contextually equivalent components that differ in their internal private state. In a fully abstract trace semantics, the trace actions must record only those heap locations that are shared between the component and the context. Consequently, the definition of the trace semantics must inductively track what has been shared in the past. In particular, the definition must account for locations reachable indirectly from explicitly shared locations. This complicates both the definition of traces and the proofs that build on the definition.

Second, the source context that the backtranslation constructs from a target trace must simulate the shared part of the heap at every context switch. Since locations in the target may be masked, the source context must maintain a map with the source locations corresponding to the target masked ones, which complicates it substantially. We call this map B. Now, this affects two patterns of target traces that need to be handled in a special way: $\texttt{call}\ \mathbf{read}\ \mathbf{v}\ \mathbf{H}?\cdot\texttt{ret}\ \mathbf{H'}!$ and $\texttt{call}\ \mathbf{write}\ \mathbf{v}\ \mathbf{H}?\cdot\texttt{ret}\ \mathbf{H'}!$. Normally, these patterns would be translated in source-level calls to the same functions (read and write), but this is not possible. In fact, the source code has no read nor write function, and the target-level calls to those functions need to be backtranslated to the corresponding source constructs ($\mathsf{!}$ and $\mathsf{:=}$, respectively). The locations used by these constructs must be looked up from B as these are reads and writes to masked locations. Moreover, calls and returns to $\mathbf{read}$ can be simply ignored since the effects of reads are already captured by later actions in traces. Calls and returns to $\mathbf{write}$ cannot be ignored as they set up a component location (albeit masked) in a certain way and that affects the behaviour of the component. We show in Example 5 how to backtranslate calls and returns to $\mathbf{write}$.

Example 5 (Backtranslation of traces)**.** Consider the trace below and its backtranslation.

$$\begin{array}{ll}
(1) & \texttt{call}\ \mathbf{f}\ \mathbf{0}\ \mathbf{1}\mapsto\mathbf{4}? \\
(2) & \texttt{ret}\ \mathbf{1}\mapsto\langle\mathbf{1},\mathbf{k_{com}}\rangle\,! \\
(3) & \left[\begin{array}{l}\texttt{call}\ \mathbf{write}\ \langle\langle\mathbf{1},\mathbf{k_{com}}\rangle\,,\mathbf{5}\rangle\ \ \mathbf{1}\mapsto\langle\mathbf{1},\mathbf{k_{com}}\rangle\,? \\ \texttt{ret}\ \mathbf{1}\mapsto\langle\mathbf{1},\mathbf{k_{com}}\rangle\,!\end{array}\right.
\end{array}$$

$$\begin{array}{l}
\mathsf{main(x)}\mapsto \\
\quad\left.\begin{array}{l}\mathsf{let\ x = new\ 4\ in\ L :: \langle x, 1\rangle} \\ \mathsf{call\ f\ 0}\end{array}\right]\ (1) \\
\quad\mathsf{let\ x = !L(1)\ in\ B :: \langle x, 1\rangle}\quad]\ (2) \\
\quad\mathsf{!B(1) := 5}\quad]\ (3)
\end{array}$$

The first action, where the context registers the first location in the list L, is as before. Then in the second action the compiled component passes to the context (in location $\mathbf{1}$) a masked location with index $\mathbf{1}$ and, later, the context writes $\mathbf{5}$ to it. The backtranslated code must recognise this pattern and store the location that, in the source, corresponds to the mask $\mathbf{1}$ in the list B (action 2). In action 3, when it is time to write $\mathsf{5}$ to that location, the code looks up the location to write to from B. ⊡

It should be clear that this proof of *FAC* is substantially harder than our corresponding proof of *RSC*, which needed neither fully abstract traces, nor tracking any mapping in the backtranslated source contexts.

15 A Fully Abstract Compiler from $\mathsf{L^U}$ to $\mathbf{L^P}$

We perform the aforementioned changes to languages.

15.1 The Source Language $\mathsf{L^U}$

In $\mathsf{L^U}$ we need to add a functionality to get the size of a heap, as that is an observable that exists in the target. In fact, in the target if one allocates something, that reveals how much it's been allocated entirely.

$$
\begin{aligned}
\textit{Components}\ \mathsf{C} &::= \overline{\mathsf{F}}; \overline{\mathsf{I}}; \overline{\mathsf{E}}\\
\textit{Exports}\ \mathsf{E} &::= \mathsf{f}\\
\textit{Expressions}\ \mathsf{e} &::= \cdots \mid \mathsf{size}
\end{aligned}
$$

$$
(\mathsf{L^U}\text{-Size})\quad \frac{\|\mathsf{H}\| = n}{\mathsf{H} \triangleright \mathsf{size} \hookrightarrow \mathsf{n}}
$$

Helpers

$$
(\mathsf{L^U}\text{-Jump-Internal})\quad \frac{\begin{gathered}((\mathsf{f'} \in \overline{\mathsf{I}} \wedge \mathsf{f} \in \overline{\mathsf{I}}) \vee\\ (\mathsf{f'} \in \overline{\mathsf{E}} \wedge \mathsf{f} \in \overline{\mathsf{E}}))\end{gathered}}{\overline{\mathsf{I}}, \overline{\mathsf{E}} \vdash \mathsf{f}, \mathsf{f'} : \mathsf{internal}}
\qquad
(\mathsf{L^U}\text{-Jump-IN})\quad \frac{\mathsf{f} \in \overline{\mathsf{I}} \wedge \mathsf{f'} \notin \overline{\mathsf{E}}}{\overline{\mathsf{I}}, \overline{\mathsf{E}} \vdash \mathsf{f}, \mathsf{f'} : \mathsf{in}}
\qquad
(\mathsf{L^U}\text{-Jump-OUT})\quad \frac{\mathsf{f} \in \overline{\mathsf{E}} \wedge \mathsf{f'} \in \overline{\mathsf{I}}}{\overline{\mathsf{I}}, \overline{\mathsf{E}} \vdash \mathsf{f}, \mathsf{f'} : \mathsf{out}}
$$

$$
(\mathsf{L^U}\text{-Plug})\quad \frac{\begin{gathered}\mathsf{A} \equiv \mathsf{H}; \overline{\mathsf{F}}\,[\cdot] \qquad \mathsf{C} \equiv \overline{\mathsf{F'}}; \overline{\mathsf{I}}; \overline{\mathsf{E}} \qquad \vdash \mathsf{C}, \overline{\mathsf{F}} : \mathsf{whole} \qquad \mathsf{main} \in \mathtt{names}(\overline{\mathsf{F}})\\ \forall \mathsf{f} \in \overline{\mathsf{E}}, \mathsf{f} \notin \mathtt{fn}(\overline{\mathsf{F}}) \qquad \forall \mathsf{f} \in \mathtt{fn}(\overline{\mathsf{F'}}), \mathsf{f} \in \overline{\mathsf{I}} \vee \mathsf{f} \in \overline{\mathsf{F'}}\end{gathered}}{\mathsf{A}\,[\mathsf{C}] = \mathsf{H}; \overline{\mathsf{F}}; \overline{\mathsf{F'}}; \overline{\mathsf{I}}; \overline{\mathsf{E}}}
$$

$$
(\mathsf{L^U}\text{-Whole})\quad \frac{\begin{gathered}\mathsf{C} \equiv \overline{\mathsf{F'}}; \overline{\mathsf{I}}; \overline{\mathsf{E}}\\ \mathtt{names}(\overline{\mathsf{F}}) \cap \mathtt{names}(\overline{\mathsf{F'}}) = \varnothing\\ \mathtt{names}(\overline{\mathsf{I}}) \subseteq \mathtt{names}(\overline{\mathsf{F}}) \cup \mathtt{names}(\overline{\mathsf{F'}})\\ \mathtt{fv}(\overline{\mathsf{F}}) \cup \mathtt{fv}(\overline{\mathsf{F'}}) = \varnothing\end{gathered}}{\vdash \mathsf{C}, \overline{\mathsf{F}} : \mathsf{whole}}
\qquad
(\mathsf{L^U}\text{-Initial State})\quad \frac{\begin{gathered}\mathsf{P} \equiv \mathsf{H}; \overline{\mathsf{F}}; \overline{\mathsf{I}}; \overline{\mathsf{E}}\\ \mathsf{C} \equiv \overline{\mathsf{F}}; \overline{\mathsf{I}}; \overline{\mathsf{E}}\end{gathered}}{\Omega_0\,(\mathsf{P}) = \mathsf{C}; \mathsf{H} \triangleright \mathsf{call\ main\ 0}}
$$

The semantics is unchanged, it only relies on the new helper functions above.

15.2 The Target Language $\mathbf{L^P}$

15.2.1 Syntax Changes

$$
\begin{aligned}
\textit{Components}\ \mathbf{C} &::= \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}}; \mathbf{k_{root}}, \mathbf{k_{com}}\\
\textit{Exports}\ \mathbf{E} &::= \mathbf{f}\\
\textit{Expressions}\ \mathbf{e} &::= \cdots \mid \mathbf{isloc(e)} \mid \mathbf{ispair(e)} \mid \mathbf{eqcap(e, e)}\\
\textit{Trace states}\ \boldsymbol{\Theta} &::= (\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{t})_{\overline{\mathbf{f}}})\\
\textit{Trace bodies}\ \mathbf{t} &::= \mathbf{s} \mid \mathbf{unk}\\
\textit{Trace labels}\ \delta &::= \epsilon \mid \beta
\end{aligned}
$$

$$\begin{aligned} \textit{Trace actions }\beta &::= \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}? \mid \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}! \mid \texttt{ret}\ \mathbf{H}! \mid \texttt{ret}\ \mathbf{H}? \mid \downarrow \mid \uparrow \mid \texttt{write}(\mathbf{v}, \mathbf{n}) \\ \textit{Traces }\overline{\beta} &::= \varnothing \mid \overline{\beta} \cdot \beta \end{aligned}$$

We assume programs are given two capabilities they own: $\mathbf{k_{root}}$ and $\mathbf{k_{com}}$ and that the attacker does not have. The former is used to create a part of the heap for component-managed datastructures. The latter does not even hide a location, we need it as a placeholder.

Traces in this case have the same syntactic structure as before, but they do not carry the whole heap. So we use a different symbol (β), to visually distinguish between the two traces and the kind of information carried by them.

We need a write label $\texttt{write}(\mathbf{v}, \mathbf{n})$ that tells that masked location $\mathbf{n}$ has been updated to value $\mathbf{v}$. This captures the usage of compiler-inserted functions to read and write masked locations (concepts that will be clear once the compiler is defined). The read label is not needed because its effect are captured anyway by call/return.

Trace states are either operational semantics states or an unknown state, mimicking the execution in a context. The former has an addtional element $\overline{\mathbf{n}}$, the list of locations shared with the context. The latter carries the information about the component and the heap comprising the one private to the component and the one shared with the context. It also carries the stack of function calls, where we add symbol $\mathbf{unk}$ to indicate when the called function was in the context.

Helper functions are as above.

15.2.2 Semantics Changes

In $\mathbf{L^P}$ we need functionality to tell if a pair is a location or not and to traverse values in order to extract such locations.

$$\frac{\begin{array}{c}(\mathbf{H} \triangleright \mathbf{e} \hookrightarrow \langle \mathbf{n}, \mathbf{v} \rangle \quad \mathbf{n} \mapsto \_ ; \eta \in \mathbf{H} \quad \eta = \mathbf{v} \text{ or } \eta = \bot) \Rightarrow \mathbf{b} = \mathbf{true} \\ \text{otherwise } \mathbf{b} = \mathbf{false}\end{array}}{\mathbf{H} \triangleright \mathbf{isloc}(\mathbf{e}) \hookrightarrow \mathbf{b}}\ (\mathbf{L^P}\text{-isloc})$$

$$\frac{\begin{array}{c}\mathbf{H} \triangleright \mathbf{e} \hookrightarrow \langle \mathbf{v}, \mathbf{v} \rangle \Rightarrow \mathbf{b} = \mathbf{true} \\ \text{otherwise } \mathbf{b} = \mathbf{false}\end{array}}{\mathbf{H} \triangleright \mathbf{ispair}(\mathbf{e}) \hookrightarrow \mathbf{b}}\ (\mathbf{L^P}\text{-ispair})$$

These are used to traverse the value stored at a location and extract all sublocations stored in there. There may be pairs containing pairs etc, and thus when we need to know if something is a pair before projecting out. Also, we need to know if a pair is a location or not, in order to know whether or not we can dereference it.

Additionally, we need a functionality to tell if two capabilities are the same. Now, this could be problematic because it could reveal capability allocation order and thus introduce observations that we do not want. However, the compiler will ensure that the context only receives $\mathbf{k_{com}}$ as a capability and never a newly-allocated capability. So the context will not be able to test equality of capabilities generated by the compiled component as it will effectively see only one.

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow \mathbf{k} \qquad \mathbf{H} \triangleright \mathbf{e'} \hookrightarrow \mathbf{k}}{\mathbf{H} \triangleright \mathbf{eqcap(e, e')} \hookrightarrow \mathbf{true}}\ (\mathbf{L^P}\text{-eqcap-true})$$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow \mathbf{k} \qquad \mathbf{H} \triangleright \mathbf{e'} \hookrightarrow \mathbf{k'} \qquad \mathbf{k} \neq \mathbf{k'}}{\mathbf{H} \triangleright \mathbf{eqcap(e, e')} \hookrightarrow \mathbf{false}}\ (\mathbf{L^P}\text{-eqcap-false})$$

15.2.3 A Fully Abstract Trace Semantics for $\mathbf{L^P}$

$\mathbf{\Theta} \xrightarrow{\beta}\!\!\twoheadrightarrow \mathbf{\Theta'}$ State $\mathbf{\Theta}$ emits visible action β becoming $\mathbf{\Theta'}$.

$\mathbf{\Theta} \overset{\overline{\beta}}{\Longrightarrow}\!\!\twoheadrightarrow \mathbf{\Theta'}$ State $\mathbf{\Theta}$ emits trace $\overline{\beta}$ becoming $\mathbf{\Theta'}$.

Helper functions

$$\frac{\begin{array}{cc}\mathbf{n} \in \mathtt{reach}(\mathbf{n_{st}}, \mathbf{k_{st}}, \mathbf{H}) & \mathbf{n_{st}} \mapsto \_ : \_ \in \mathbf{H'} \\ \mathbf{k_{st}} \in \mathbf{k_{root}} \cup \mathbf{H'} & \mathbf{n} \mapsto \mathbf{v} : \eta \in \mathbf{H}\end{array}}{\mathbf{H} \vdash \mathtt{reachable}(\mathbf{n}, \mathbf{H'})}\ (\text{Reachable}) \qquad \frac{\forall \mathbf{k} \in \mathbf{H}.\ \mathbf{k} \notin \mathbf{v}}{\vdash \mathtt{valid}(\mathbf{v}, \mathbf{H})}\ (\text{Valid value})$$

$$\frac{\begin{array}{c}\mathbf{H} = \mathbf{H_{priv}} \cup \mathbf{H_{sha}} \qquad \mathbf{H'} = \mathbf{H_{priv}} \cup \mathbf{H'_{sha}} \cup \mathbf{H_{new}} \\ \mathbf{H''} = \mathbf{H'_{sha}} \cup \mathbf{H_{new}} \qquad \mathtt{dom}(\mathbf{H}) = \overline{\mathbf{n}} \qquad \mathtt{dom}(\mathbf{H''}) = \overline{\mathbf{n'}} \\ \mathbf{k} \in \mathbf{H_{sha}} \iff \mathbf{k} \in \mathbf{H'_{sha}} \\ \forall \mathbf{k'} \in \mathbf{H_{new}}.\ \mathbf{k'} \notin \mathbf{H_{priv}} \cup \mathbf{H_{sha}} \\ \forall \mathbf{n} \mapsto \mathbf{v}; \eta \in \mathbf{H_{sha}}.\ \mathbf{n} \mapsto \mathbf{v''}; \eta \in \mathbf{H'_{sha}} \wedge \vdash \mathtt{valid}(\mathbf{v''}, \mathbf{H}) \\ \forall \mathbf{n'} \mapsto \mathbf{v'} : \eta' \in \mathbf{H_{new}}.\ \vdash \mathtt{valid}(\mathbf{v'}, \mathbf{H_{priv}} \cup \mathbf{H'_{sha}}) \wedge \\ \mathbf{H''} \vdash \mathtt{reachable}(\mathbf{v'}, \mathbf{H_{priv}} \cup \mathbf{H'_{sha}})\end{array}}{\vdash \mathtt{validHeap}(\mathbf{H}, \mathbf{H'}, \mathbf{H''}, \overline{\mathbf{n}}, \overline{\mathbf{n'}})}\ (\text{Valid heap})$$

$\boxed{\mathbf{\Theta} \xrightarrow{\overline{\beta}}\!\!\twoheadrightarrow \mathbf{\Theta'}}$

$$\frac{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \xrightarrow{\epsilon} (\mathbf{C}; \mathbf{H'}; \overline{\mathbf{n}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}})}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \xrightarrow{\epsilon}\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H'}; \overline{\mathbf{n}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}})}\ (\mathbf{L^P}\text{-Traces-Silent})$$

$$\frac{\begin{array}{c}\mathbf{C} = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}} \qquad \mathbf{f} \in \overline{\mathbf{E}} \qquad \mathbf{f(x)} \mapsto \mathbf{s; return;} \in \overline{\mathbf{F}} \\ \overline{\mathbf{f'}} = \overline{\mathbf{f}} \cdot \mathbf{f} \qquad \mathbf{H} \vdash \mathtt{valid}(\mathbf{v}) \\ \vdash \mathtt{validHeap}(\mathbf{H}, \mathbf{H''}, \mathbf{H'}, \overline{\mathbf{n}}, \overline{\mathbf{n'}})\end{array}}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f}}}) \xrightarrow{\mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H'}?}\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H''}; \overline{\mathbf{n'}} \triangleright (\mathbf{s; return;})_{\overline{\mathbf{f'}}})}\ (\mathbf{L^P}\text{-Traces-Call})$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Returnback)}\quad \frac{\begin{array}{c}\overline{\mathbf{f}} = \overline{\mathbf{f'}} \cdot \mathbf{f} \\ \vdash \texttt{validHeap}(\mathbf{H}, \mathbf{H''}, \mathbf{H'}, \overline{\mathbf{n}}, \overline{\mathbf{n'}})\end{array}}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f}}}) \xrightarrow{\texttt{ret ?}\mathbf{H'}}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H''}; \overline{\mathbf{n'}} \triangleright (\mathbf{skip})_{\overline{\mathbf{f'}}})}$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Callback)}\quad \frac{\begin{array}{ccc}\mathbf{s} = \mathbf{call\ f\ e} & & \mathbf{H} \triangleright \mathbf{e} \hookrightarrow \mathbf{v} \\ \mathbf{C} = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}} & \overline{\mathbf{f'}} = \overline{\mathbf{f}} \cdot \mathbf{f} & \mathbf{f} \in \overline{\mathbf{I}} \\ \overline{\mathbf{n}} \subseteq \overline{\mathbf{n'}} = \{\mathbf{n} \mid \mathbf{H} \vdash \texttt{reachable}(\mathbf{n}, \mathbf{H})\} & & \mathbf{H'} = \mathbf{H}|_{\overline{\mathbf{n'}}}\end{array}}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \xrightarrow{\texttt{call f v }\mathbf{H'}\texttt{!}}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H}; \overline{\mathbf{n'}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f'}}})}$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Return)}\quad \frac{\begin{array}{ccc}\mathbf{C} = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}} & \overline{\mathbf{f}} = \overline{\mathbf{f'}} \cdot \mathbf{f} & \mathbf{f} \in \overline{\mathbf{E}} \\ \overline{\mathbf{n}} \subseteq \overline{\mathbf{n'}} = \{\mathbf{n} \mid \mathbf{H} \vdash \texttt{reachable}(\mathbf{n}, \mathbf{H})\} & & \mathbf{H'} = \mathbf{H}|_{\overline{\mathbf{n'}}}\end{array}}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{return;})_{\overline{\mathbf{f}}}) \xrightarrow{\texttt{ret !}\mathbf{H'}}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H}; \overline{\mathbf{n'}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f'}}})}$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Terminate)}\quad \frac{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \overset{\epsilon}{\not\rightarrow} \_}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \xrightarrow{\downarrow}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}})}$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Diverge)}\quad \frac{\forall n.\ (\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \xrightarrow{\epsilon}{}^{\mathbf{n}} (\mathbf{C}; \mathbf{H'}; \overline{\mathbf{n'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}})}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}) \xrightarrow{\uparrow}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}})}$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Write)}\quad \frac{\begin{array}{c}\mathbf{C} = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}} \qquad \mathbf{write} \in \overline{\mathbf{E}} \qquad \mathbf{write}(\mathbf{x}) \mapsto \mathbf{s}; \mathbf{return;} \in \overline{\mathbf{F}} \\ \mathbf{C}; \mathbf{H} \triangleright \mathbf{s}[\mathbf{n} \,/\, \mathbf{x}]; \mathbf{return;} \longrightarrow * \mathbf{C}; \mathbf{H'} \triangleright \mathbf{return;}\end{array}}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f}}}) \xrightarrow{\texttt{write}(\mathbf{v},\mathbf{n})}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H'}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f}}})}$$

$$\textsf{(}\mathbf{L^{P}}\textsf{-Traces-Read)}\quad \frac{\begin{array}{c}\mathbf{C} = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}} \qquad \mathbf{read} \in \overline{\mathbf{E}} \qquad \mathbf{read}(\mathbf{x}) \mapsto \mathbf{s}; \mathbf{return;} \in \overline{\mathbf{F}} \\ \mathbf{C}; \mathbf{H} \triangleright \mathbf{s}; \mathbf{return;} \longrightarrow * \mathbf{C}; \mathbf{H'} \triangleright \mathbf{return;}\end{array}}{(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f}}}) \xrightarrow{\epsilon}\!\!\!\!\twoheadrightarrow (\mathbf{C}; \mathbf{H'}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\overline{\mathbf{f}}})}$$

$$\boxed{\boldsymbol{\Theta} \overset{\overline{\beta}}{\Longrightarrow\!\!\!\!\Rightarrow} \boldsymbol{\Theta'}}$$

$$\textsf{(E}\mathbf{L^{P}}\textsf{-single)}\quad \frac{\boldsymbol{\Omega} \Longrightarrow \boldsymbol{\Omega''} \qquad \boldsymbol{\Omega''} \xrightarrow{\beta} \boldsymbol{\Omega'}}{\boldsymbol{\Omega} \xRightarrow{\beta} \boldsymbol{\Omega'}} \qquad \textsf{(E}\mathbf{L^{P}}\textsf{-silent)}\quad \frac{\boldsymbol{\Omega} \xrightarrow{\epsilon} \boldsymbol{\Omega'}}{\boldsymbol{\Omega} \Longrightarrow \boldsymbol{\Omega'}}$$

$$\textsf{(E}\mathbf{L^{P}}\textsf{-trans)}\quad \frac{\boldsymbol{\Omega} \xRightarrow{\overline{\beta}} \boldsymbol{\Omega''} \qquad \boldsymbol{\Omega''} \xRightarrow{\overline{\beta'}} \boldsymbol{\Omega'}}{\boldsymbol{\Omega} \xRightarrow{\overline{\beta} \cdot \overline{\beta'}} \boldsymbol{\Omega'}}$$

$$\frac{\mathbf{n} \in \overline{\mathbf{n}} \iff \mathbf{n} \mapsto \mathbf{v}; \eta \in \mathbf{H} \qquad \mathbf{main} \notin \mathtt{dom}(\overline{\mathbf{F}}) \qquad \mathbf{C} = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{E}}}{\mathbf{\Theta_0}\left(\mathbf{C}\right) = \left(\mathbf{C}; \mathbf{H}; \overline{\mathbf{n}} \triangleright (\mathbf{unk})_{\mathbf{main}}\right)} \; (\mathbf{L^P}\text{-Traces-Initial})$$

$$\mathsf{TR}(\mathbf{C}) = \left\{ \overline{\beta} \;\middle|\; \mathbf{\Theta_0}\left(\mathbf{C}\right) \xRightarrow{\overline{\beta}} \_ \right\}$$

15.2.4 Results about the Trace Semantics

The following results hold for $\mathbf{C_1} = \wr \mathsf{C_1} \int^{\mathsf{L^U}}_{\mathbf{L^P}}$ and $\mathbf{C_2} = \wr \mathsf{C_2} \int^{\mathsf{L^U}}_{\mathbf{L^P}}$.

Property 1 (Heap locations)**.** AS mentioned, the trace semantics carries the whole shared heap: locations created by the compiled component and then passed to the context and locations created by the context and passed to the compiled component. We can really partition the heap as follows then:

location \creator	$\wr \mathsf{C} \int^{\mathsf{L^U}}_{\mathbf{L^P}}$	$\mathbb{C}$
private	(1) to $\wr \mathsf{C} \int^{\mathsf{L^U}}_{\mathbf{L^P}}$	(2) to $\mathbb{C}$
shared	(3) with $\mathbb{C}$	(4) with $\wr \mathsf{C} \int^{\mathsf{L^U}}_{\mathbf{L^P}}$

Now, for compiled components there never are locations of kind 3. That is because those locations are masked and never passed, never made accessible to the context. So really, the trace semantics only collects locations of kind 4 on the traces.

Lemma 27 (Correctness)**.**

$$\text{if } \mathbf{C_1} \simeq_{ctx} \mathbf{C_2}$$
$$\text{then } \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$$

Proof Sketch. By contraposition:

$$\text{if } \mathsf{TR}(\mathbf{C_1}) \neq \mathsf{TR}(\mathbf{C_2})$$
$$\text{then } \exists \mathbf{A}.\ \mathbf{A}\left[\mathbf{C_1}\right]^{\Downarrow} \wedge \mathbf{A}\left[\mathbf{C_2}\right]{\Uparrow}(wlog)$$

We are thus given $\overline{\beta_1} = \overline{\beta} \cdot \beta_1$ and $\overline{\beta_2} = \overline{\beta} \cdot \beta_2$ and $\beta_1 \neq \beta_2$.

We can construct a context $\mathbf{A}$ that replicates the behaviour of $\overline{\beta}$ and then performs the differentiation.

This is a tedious procedure that is analogous to existing results [9, 15] and analogous to the backtranslation of Section 4.2.

The actions only share the heap that is reachable from both sides, the heap that is private to the component is never touched, so reconstructing the heap is possible. The reachability conditions on the heap also ensure this.

The differentiation is based on differences on the actions which are visible and reachable, so that is also possible. □

Lemma 28 (Completeness)**.**

$$\begin{aligned} &\text{if } \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2}) \\ &\text{then } \mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \end{aligned}$$

Proof Sketch. By contradiction let us assume that

$$\exists \mathbf{A}.\ \mathbf{A}\left[\mathbf{C_1}\right]\Downarrow \wedge \mathbf{A}\left[\mathbf{C_2}\right]\Uparrow (wlog)$$

Contexts are deterministic, so they cannot behave differently based on the values of locations that are never shared with $\mathbf{C_1}$ or $\mathbf{C_2}$.

The semantics forbids guessing, so a context will never have access to the locations that $\mathbf{C_1}$ or $\mathbf{C_2}$ do not share.

Thus a context can exhibit a difference in behaviour by relying on something that $\mathbf{C_1}$ modified unlike $\mathbf{C_2}$ and that can be:

- a parameter passed in a call.

  This contradicts the hypothesis that the trace semantics is the same as that parameter is captured in the $\mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ label.

- the value of a shared location.

  This contradicts the hypothesis that the trace semantics is the same as all locations that are reachable both by the context and by $\mathbf{C_1}$ and $\mathbf{C_2}$ are captured on the labels

Having reached a contradiction, this case holds. □

Lemma 29 (Full abstraction of the trace semantics for compiled components)**.**

$$\mathsf{TR}(\llbracket \mathsf{C}_1 \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}) = \mathsf{TR}(\llbracket \mathsf{C}_2 \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}) \iff \llbracket \mathsf{C}_1 \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} \simeq_{ctx} \llbracket \mathsf{C}_2 \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$$

Proof. By Lemmas 27 and 28. □

15.3 The Compiler $\llbracket \cdot \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}$

$$\begin{aligned} \llbracket \overline{\mathsf{F}}; \overline{\mathsf{I}}; \overline{\mathsf{E}} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}} = &\llbracket \overline{\mathsf{F}} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}, \mathbf{read(x)} \mapsto \mathbf{s_{read}}, \mathbf{write(x)} \mapsto \mathbf{s_{write}}, \\ &\ \mathbf{mask(x)} \mapsto \mathbf{s_{mask}}, \mathbf{unmask(x)} \mapsto \mathbf{s_{unmask}}, \cdots; \\ &\llbracket \overline{\mathsf{I}} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}; \\ &\llbracket \overline{\mathsf{E}} \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}, \mathbf{read}, \mathbf{write}; \\ &\mathbf{k_{root}}, \mathbf{k_{com}} \end{aligned} \qquad (\llbracket \cdot \rrbracket^{\mathsf{L^U}}_{\mathbf{L^P}}\text{-Comp})$$

$$\wr \mathsf{f(x) \mapsto s; return;} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{f(x) \mapsto s_{add}(x);}\ \mathbf{s_{pre};}\ [\![\mathsf{s}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}};\ \mathbf{s_{post};}\ \mathbf{return;} \qquad (\wr\cdot\int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\text{-Function})$$

$$\wr \mathsf{f} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{f} \qquad (\wr\cdot\int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\text{-Interfaces})$$

$$\wr \mathsf{f} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{f} \qquad (\wr\cdot\int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\text{-Exports})$$

Expression translation unmodified: $\wr \mathsf{e} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = [\![\mathsf{e}]\!]^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}$

Statement translation unmodified except for

$$\wr \mathsf{let\ x = new\ e\ in\ s} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{let\ x_{loc} = new}\ \wr \mathsf{e} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\ \mathbf{in}\ \mathbf{let\ x_{cap} = hide\ x_{loc}\ in}\ \mathbf{s_{register}(x_{loc}, x_{cap});}\ \mathbf{let\ x = \langle x_{loc}, x_{cap}\rangle\ in}\ \wr \mathsf{s} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} \qquad (\wr\cdot\int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\text{-New})$$

$$\wr \mathsf{call\ f\ e} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}} = \mathbf{let\ x} = \wr \mathsf{e} \int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\ \mathbf{in\ s_{add}(x); s_{post}; call\ f\ x; s_{pre}} \qquad (\wr\cdot\int^{\mathsf{L}^{\mathsf{U}}}_{\mathbf{L}^{\mathbf{P}}}\text{-Call})$$

So the compiler is mostly unchanged.

The compiled code will maintain the following invariant:

- no locations (even though protected by capabilities) are ever made accessible "in clear" to the context;
- "made accessible" means either passed as a parameter or through a shared location;
- instead, before passing control to the context, all component-created locations that are shared with the context are masked, i.e., their representation $\langle \mathbf{n}, \mathbf{k}\rangle$ is replaced with $\langle \mathbf{n}', \mathbf{k_{com}}\rangle$, where $\mathbf{n}'$ is their index in the list of shared locations that the compiled component keeps.
- when receiving control from the context, the compiled component ensures that all component-created locations that are shared are unmasked, i.e., upon regaining control the component replaces all values $\langle \mathbf{n}', \mathbf{k_{com}}\rangle$ that are sub-values of reachable locations with $\langle \mathbf{n}, \mathbf{k}\rangle$, which is the $\mathbf{n}'$th pair in the list of component-allocated locations;
- what is a "component-shared" location? A shared location is a pair $\langle \mathbf{n}, \mathbf{k}\rangle$ where (i) $\mathbf{k}$ is a capability created by the compiled component and (ii) the pair is stored in the heap at a location that the context can dereference (perhaps not directly).
- In order to define what is a shared location, the compiled component keeps a list of all the locations that have been passed to it and that the context

created. These locations can only be in $\langle \mathbf{n}, \_ \rangle$ form, where $\_$ is either a capability or not depending whether the context hid the location. These locations can only be pairs since we know that a compiled component will only use pairs as locations, mimicking the source semantics.

We normally do not know what locations will be accessed, but given a parameter that is a location, we can scan its contents to understand what new locations are passed.

- The compiled component thus can keep a list of "shared" locations: those whose contents are accessible both by the context and by itself. These locations created by the context are acquired as parameters or as locations reachable by a parameter. These locations created by the component are tracked as those hidden with a component-created capability and reachable from a shared location.
- The only concern that can arise is if we create location $\mathbf{n}$ and then add it to the list of shared locations at index $\mathbf{n'}$. That location $\langle \mathbf{n}, \mathbf{k} \rangle$ would be masked as $\langle \mathbf{n'}, \mathbf{k_{com}} \rangle$, which grants the context direct access to it. This is where we need to use $\mathbf{k_{com}}$ as leaking different capabilities would lead to differentiation between components. Fortunately, the context starts execution and, in order to call the compiled component, it must allocate at least one location, so this problem cannot arise.

15.3.1 Syntactic Sugar

The languages we have do not let us return directly a value. In the following however, for readability, we write

$$\mathbf{let\ x = func\ v\ in\ s}$$

to intend: call function $\mathbf{func}$ with parameter $\mathbf{v}$ and store its returned value in $\mathbf{x}$ for use in $\mathbf{s}$.

We indicate how that statement can be expressed in our language with the following desugaring:

$$\mathbf{let\ y = new\ 0\ in\ let\ z = \langle v, y \rangle\ \ in\ call\ func\ z; let\ x = !z.2\ with\ 0\ in\ s}$$

15.3.2 Support Data Structures

The compiler relies on a number of data structures it keeps starting from location $\mathbf{0}$, which is accessible via $\mathbf{k_{root}}$.

These data structures are:

- a list of capabilities, which we denote with $\overline{\mathbf{K}}$. These capabilities are those that the compiled component has allocated.
- a list of component-allocated locations, which we denote with $\overline{\mathbf{L}}$. These are locations $\langle \mathbf{n}, \mathbf{k} \rangle$ that are created by the compiled component and whose $\mathbf{k}$ are elements of $\overline{\mathbf{K}}$

- a list of shared locations, which we denote with $\overline{\mathbf{S}}$. These are either (i) locations that are created by the context and passed to the compiled component or (ii) locations that are created by the compiled component and passed to the context.

Given a list L of elements e, we use these helper functions:

- $\texttt{indexof}(L, e)$ returns n, the index of e in L, or 0 if e is not in L;
- $L(n)$ returns the nth element e of L or 0 if the list length is shorter than n;
- $L :: e$ if e is not in L, it adds element e to the list, increasing its length by 1;
- $\texttt{rem}(L, e)$ removes element e from L;
- $e \in L$ returns true or false depending on whether e is in L or not.

We keep this abstract syntax for handling lists and do not write the necessary recursive functions as they would only be tedious and hardly readable. Realistically, we would also need a temporary list for accumulating results etc, again, this is omitted for simplicity and readability.

15.3.3 Support Functions

Read

$$\begin{aligned}
\mathbf{s_{read}} = {} & \mathbf{let\ x_n{=}x.1.1\ in} \\
& \quad \mathbf{let\ x_k{=}x.1.2\ in} \\
& \quad\ \mathbf{let\ x_{real}{=}\overline{L}(x_n)\ in} \\
& \quad\quad \mathbf{let\ x_{dest}{=}x.2.1\ in} \\
& \quad\quad\ \mathbf{let\ x_{dcap}{=}x.2.2\ in} \\
& \quad\quad\quad \mathbf{let\ x_{val}{=}!x_{real}\ with\ x_k\ in} \\
& \quad\quad\quad\ \mathbf{x_{dest} := x_{val}\ with\ x_{dcap}}
\end{aligned}$$

In order to read a location $\langle \mathbf{n}, \mathbf{k} \rangle$, we receive that as the first projection of parameter $\mathbf{x}$. Because we do not explicitly return values, we need the second projection of $\mathbf{x}$ to contain the destination where to target receives the result of the read.

We split the pair in the masking index $\mathbf{x_n}$ and in the capability to access the location $\mathbf{x_k}$. Then we lookup the location in the list of component-created locations and return its value. We do not need to mask its contents as we know that they have already been masked when this location was shared with the context (line 5 of the postamble). We do not need to add its contents to the list of shared locations as that is already done in lines 2 and 3 of the postamble.

Write

$$
\begin{aligned}
\mathbf{s_{write}} = {} & \mathbf{let}\ \mathbf{x_n}{=}\mathbf{x.1.1}\ \mathbf{in}\\
& \mathbf{let}\ \mathbf{x_k}{=}\mathbf{x.1.2}\ \mathbf{in}\\
& \mathbf{let}\ \mathbf{x_{real}}{=}\overline{\mathbf{L}}(\mathbf{x_n})\ \mathbf{in}\\
& \mathbf{x_{real}} := \mathbf{x.2}\ \mathbf{with}\ \mathbf{x_k};
\end{aligned}
$$

In order to write value $\mathbf{v}$ a location $\langle \mathbf{n}, \mathbf{k}\rangle$ we receive a parameter structured as follows: $\mathbf{x} \equiv \langle \mathbf{n}, \mathbf{k}\rangle , \mathbf{v}$. Then we unfold the elements of the parameter and lookup element $\mathbf{n}$ in the list of component-defined locations. We use this looked-up element to write the value $\mathbf{v}$ there.

We do not need to mask $\mathbf{v}$ because it cannot point to locations that are created by the compiled component.

At this stage, $\mathbf{v}$ may contain new locations created by the context and that are now shared. We do not add them now to the list of shared locations because we know that upon giving control again to the compiled component, the preamble will do this.

Mask

$$
\begin{aligned}
\mathbf{s_{mask}} = {} & \forall\langle \mathbf{n}, \mathbf{k}\rangle \in \mathbf{x}.\ \mathbf{isloc}(\langle \mathbf{n}, \mathbf{k}\rangle)\\
& \mathbf{if}\ \mathbf{k} \in \overline{\mathbf{K}}\\
& \mathbf{replace}\ \langle \mathbf{n}, \mathbf{k}\rangle\ \mathbf{with}\ \left\langle \mathtt{indexof}(\overline{\mathbf{L}}, \mathbf{n}), \mathbf{k_{com}}\right\rangle
\end{aligned}
$$

We use the abstract construct **replace...** to indicate the following. We want to keep the value passed as parameter $\mathbf{x}$ unchanged but replace its subvalues that are pairs and, more specifically, component-created locations, with a pair with its location masked to be the index in the list of component-allocated locations.

This can be implemented by checking the sub-values of a value via the **ispair** and **isloc** expressions, we omit its details for simplicity. To ensure $\in \overline{\mathbf{K}}$ is implementable, we use the **eqcap** expression.

Masked locations cannot mention their capability or they would leak this information and generate different traces for equivalent compiled programs.

Unmask

$$
\begin{aligned}
\mathbf{s_{unmask}} = {} & \forall\langle \mathbf{n}, \mathbf{k}\rangle \in \mathbf{x}\\
& \mathbf{if}\ \mathbf{k} == \mathbf{k_{com}}\\
& \mathbf{replace}\ \langle \mathbf{n}, \mathbf{k}\rangle\ \mathbf{with}\ \overline{\mathbf{L}}(\mathbf{n})
\end{aligned}
$$

In the case of unmasking, we receive a value through parameter $\mathbf{x}$ and we know that there may be subvalues of it of the form $\langle \mathbf{n}, \mathbf{k}\rangle$ where $\mathbf{n}$ is an index in the component-created shared locations. So we lookup the element from that list and replace it in $\mathbf{x}$.

15.3.4 Inlined Additional Statements (Preamble, Postamble, etc)

Adding

$$\begin{aligned}
\mathbf{s_{add}}(\mathbf{x}) = {} & \mathbf{if\ isloc}(\mathbf{x})\ \mathbf{then} \\
& \overline{\mathbf{S}} :: \mathbf{x}; \\
& \mathbf{if\ x.2} \in \overline{\mathbf{K}}\ \mathbf{then}\ \overline{\mathbf{L}} :: \mathbf{x\ else\ skip}
\end{aligned}$$

This common part ensures that the parameter $\mathbf{x}$ is added to the list of shared locations (line 1) and then, if the capability is locally-created, it is also added to the list of locally-shared locations (line 2).

The second line is for when this code is called before a $\wr\mathsf{call\ f}\int^{\mathsf{L^U}}_{\mathbf{L^P}}$.

Registration

$$\mathbf{s_{register}}(\mathbf{x_{loc}}, \mathbf{x_{cap}}) = \overline{\mathbf{K}} :: \mathbf{x_{cap}};$$

This statement registers capability $\mathbf{x_{cap}}$ in the list of component-created capabilities.

Preamble The preamble is responsible of adding all context-created locations to the list of shared locations and to ensure that all contents of shared locations are unmasked, as the compiled code will operate on them.

$$\begin{aligned}
\mathbf{s_{pre}} = {} & \forall\langle\mathbf{n},\mathbf{k}\rangle \in \mathtt{reach}(\overline{\mathbf{S}}).\ \mathbf{isloc}(\langle\mathbf{n},\mathbf{k}\rangle) \\
& \quad \mathbf{if}\ \langle\mathbf{n},\mathbf{k}\rangle \notin \overline{\mathbf{S}}\ \mathbf{then}\ \overline{\mathbf{S}} :: \langle\mathbf{n},\mathbf{k}\rangle\,;\ \mathbf{else\ skip} \\
& \quad \forall\langle\mathbf{n},\mathbf{k}\rangle \in \overline{\mathbf{S}}.\ \mathbf{isloc}(\langle\mathbf{n},\mathbf{k}\rangle) \\
& \quad\ \ \mathbf{let\ x = unmask(!n\ with\ k)\ in\ n := x\ with\ k}
\end{aligned}$$

First any location that is reachable from the shared locations (line 1) and that is not a shared location already is added to the list of shared locations (line 2). By where this code is placed we know that these new locations can only be context-created.

Then, for all shared locations (line 3), we unmask their contents using the $\mathbf{unmask}$ function (line 4).

Postamble The postamble is responsible of adding all component-created locations to the list of shared locations and of component-created shared locations and to ensure that all shared locations are masked as the context will operate on them.

$$\begin{aligned}
\mathbf{s_{post}}(\mathbf{x}) = {} & \forall\langle\mathbf{n},\mathbf{k}\rangle \in \mathtt{reach}(\overline{\mathbf{S}}).\ \mathbf{isloc}(\langle\mathbf{n},\mathbf{k}\rangle) \\
& \quad \mathbf{if}\ \langle\mathbf{n},\mathbf{k}\rangle \notin \overline{\mathbf{S}}\ \mathbf{then}\ \overline{\mathbf{S}} :: \langle\mathbf{n},\mathbf{k}\rangle\,; \overline{\mathbf{L}} :: \langle\mathbf{n},\mathbf{k}\rangle\,;\ \mathbf{else\ skip} \\
& \forall\langle\mathbf{n},\mathbf{k}\rangle \in \overline{\mathbf{S}}.\ \mathbf{isloc}(\langle\mathbf{n},\mathbf{k}\rangle) \\
& \quad \mathbf{let\ x = mask(!n\ with\ k)\ in\ n := x\ with\ k}
\end{aligned}$$

Then for all locations that are reachable from a shared location (line 1), and that are not already there (line 2), we add those locations to the list of shared locations and to the list of component-created shared locations (line 2). Then for all shared locations (line 3), we mask their contents using the **mask** function (line 4).

15.4 The Trace-based Backtranslation: $\langle\!\langle \boxed{\cdot} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$

Value backtranslation is the same, so $\langle\!\langle \boxed{\mathbf{v}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}} = \langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$.

15.4.1 The Skeleton

The skeleton is almost as before (Section 4.2.2), with the only addition of another list B explained below.

The only additions are two functions terminate and diverge, which do what their name suggests:

$$\mathsf{terminate(x)} \mapsto \mathsf{fail}$$
$$\mathsf{diverge(x)} \mapsto \mathsf{call\ diverge\ 0}$$

15.4.2 The Common Prefix

$\texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}?$ As in Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-call), we keep a list of the context-allocated locations and we update them. Also, we extend that list.

$\texttt{ret}\ ?\mathbf{H}$ As above.

$\texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ This is analogous to Rule ($\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$-callback-loc) but with a major complication.

Now this is complex because in the target we don't receive locations $\langle \mathbf{n}, \mathbf{k} \rangle$ from the compiled component, but masked indices $\langle \mathbf{i}, \mathbf{k_{com}} \rangle$. (using $\mathbf{i}$ as a metavariable for natural numbers outputted by the masking function) We need to extract them based on where they are located in memory, knowing that the same syntactic structure is maintained in the source. So what before was relying on the relation on values $\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle$ now is no longer true because we have $\ell \approx_\beta \langle \mathbf{i}, \mathbf{k_{com}} \rangle$ which cannot hold. We need to keep a this relation as a runtime argument in the backtrnanslation and base it solely on the syntactic occurrencies of $\langle \mathbf{i}, \mathbf{k_{com}} \rangle$. So this runtime relation maps target masking indices to source locations.

So this relation is really a list B where each entry has the form $\left\langle \langle\!\langle \boxed{\mathbf{i}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}, \ell \right\rangle$.

Intuitively, consider heap $\mathbf{H}$ from the action. For all of its content $\mathbf{n} \mapsto \mathbf{v} : \eta$, we do a structural analysis of $\mathbf{v}$. This happens at the meta-level, in the backtranslation algorithm. $\mathbf{v}$ may contain subvalues of the form $\langle \mathbf{i}, \mathbf{k_{com}} \rangle$, and accessing this subvalue we know is a matter of $\cdot.\mathbf{1}$ etc. So we produce an expression e with the same instructions ($\cdot.1$ etc) in the source in order to scan *at runtime* the heap H we receive after the callback is done. (so after the action here is executed and where backtranslation code executes)

Given that expression e evaluate to location ℓ, we now need to add to B the pair $\langle \mathsf{i}, \ell \rangle$ (also given that $\mathsf{i}{=}\left\langle\!\!\left\langle \boxed{\mathbf{i}} \right\rangle\!\!\right\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$).

$\texttt{ret}\ !\mathbf{H}$ As above.

$\texttt{write}(\mathbf{v}, \mathbf{i})$ In this case we need to make use of the runtime-kept relation B. We need to know what source location ℓ corresponds to $\mathbf{i}$ so we can produce the correct code: $\ell := \langle\!\langle \boxed{\mathbf{v}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}$.

ℓ is looked up as $\mathsf{B}(\left\langle\!\!\left\langle \boxed{\mathbf{i}} \right\rangle\!\!\right\rangle^{\mathbf{L^P}}_{\mathsf{L^U}})$.

15.4.3 The Differentiator

The differentiator needs to put the right code at the right place. The backtranslation already carries all necessary information to know what the right place is, this is as in previous work: the index of the action i (at the meta level) stored in location ℓ_i (at runtime) and the call stack $\overline{\mathsf{f}}$

We now go over the various cases of trace difference and see that the differentiation code exists. We consider α_1 to be the last action in the trace of $[\![\mathsf{C}_1]\!]^{\mathsf{S}}_{\mathbf{T}}$ while α_2 is the last one of $[\![\mathsf{C}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$, both made after a common prefix.

$\alpha_1 = \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ **and** $\alpha_2 = \texttt{call}\ \mathbf{g}\ \mathbf{v}\ \mathbf{H}!$ Code $\mathsf{if}\ !\ell_\mathsf{i} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{terminate}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed in the body of f while the code $\mathsf{if}\ !\ell_\mathsf{i} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{diverge}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed in the body of g.

$\alpha_1 = \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ **and** $\alpha_2 = \texttt{call}\ \mathbf{f}\ \mathbf{w}\ \mathbf{H}!$ Code

$\mathsf{if}\ !\ell == \mathsf{i}\ \mathsf{then}$

$\mathsf{if}\ \mathsf{x} == \langle\!\langle \boxed{\mathbf{v}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\ \mathsf{then}\ \mathsf{call}\ \mathsf{terminate}\ 0\ \mathsf{else}\ \mathsf{call}\ \mathsf{diverge}\ 0\ \mathsf{else}\ \mathsf{skip}$

is placed in f.

$\alpha_1 = \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ **and** $\alpha_2 = \texttt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}'!$ Here few cases can arise, consider $\mathbf{H} = \mathbf{H_1}, \mathbf{n} \mapsto \mathbf{v} : \eta, \mathbf{H_2}$ and $\mathbf{H}' = \mathbf{H_1}, \mathbf{n}' \mapsto \mathbf{v}' : \eta', \mathbf{H}'_\mathbf{2}$:

$\mathbf{v} \neq \mathbf{v}'$ We use shortcut $\mathsf{L_{glob}}(\mathsf{n})$ to indicate the location bound to name n in the list of shared locations (same as in Section 4.2.3).

Code

$\mathsf{if}\ !\ell_\mathsf{i} == \mathsf{i}\ \mathsf{then}$

$\mathsf{let}\ \mathsf{x}{=}\mathsf{L_{glob}}(\langle\!\langle \boxed{\mathbf{n}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}})\ \mathsf{in}$

$\mathsf{if}\ \mathsf{x} == \langle\!\langle \boxed{\mathbf{v}} \rangle\!\rangle^{\mathbf{L^P}}_{\mathsf{L^U}}\ \mathsf{then}\ \mathsf{call}\ \mathsf{terminate}\ 0\ \mathsf{else}\ \mathsf{call}\ \mathsf{diverge}\ 0$

$\mathsf{else}\ \mathsf{skip}$

is placed in the body of f.

$\mathbf{n} \neq \mathbf{n'}$ In this case one of the two addresses must be bigger than the other. Wlog, let's consider $\mathbf{n} = \mathbf{n'} + \mathbf{1}$.

So $\mathbf{H_1} = \mathbf{H'_1}, \mathbf{n'} \mapsto \mathbf{v'}; \eta'$ and $\mathbf{H'_2} = \varnothing$ (otherwise we'd have a binding for $\mathbf{n}$ there).

The code in this case must access the location related to $\mathbf{n}$, it will get stuck in one case and succeed in the other:

$\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{let}\ \mathsf{x} = \mathsf{update}(\langle\langle \boxed{\mathbf{n}} \rangle\rangle_{\mathsf{L^U}}^{\mathbf{L^P}}, 0)\ \mathsf{in}\ \mathsf{call}\ \mathsf{diverge}\ 0\ \mathsf{else}\ \mathsf{skip}$

$\eta \neq \eta'$ Two cases arise:

- the location is context-created: in this case the tag must be the same, so we have a contradiction;
- the location is component-created, but in this case we know that no such location is ever passed to the context (see Property 1), so we have a contradiction.

$\alpha_1 = \mathtt{ret}\ \mathbf{H}!$ **and** $\alpha_2 = \mathtt{ret}\ !$ As above.

$\alpha_1 = \mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ **and** $\alpha_2 = \mathtt{ret}\ !$ Code $\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{terminate}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed at f while $\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{diverge}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed at the top of $\bar{\mathsf{f}}$.

$\alpha_1 = \mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ **and** $\alpha_2 = \downarrow$ Code $\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{diverge}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed at f.

$\alpha_1 = \mathtt{call}\ \mathbf{f}\ \mathbf{v}\ \mathbf{H}!$ **and** $\alpha_2 = \uparrow$ Code $\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{terminate}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed at f.

$\alpha_1 = \mathtt{ret}\ \mathbf{H}!$ **and** $\alpha_2 = \downarrow$ Code $\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{diverge}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed at the top of $\bar{\mathsf{f}}$.

$\alpha_1 = \mathtt{ret}\ \mathbf{H}!$ **and** $\alpha_2 = \uparrow$ Code $\mathsf{if}\ !\ell_{\mathsf{i}} == \mathsf{i}\ \mathsf{then}\ \mathsf{call}\ \mathsf{terminate}\ 0\ \mathsf{else}\ \mathsf{skip}$ is placed at the top of $\bar{\mathsf{f}}$.

$\alpha_1 = \downarrow$ **and** $\alpha_2 = \uparrow$ Nothing to do, the compiled component performs the differentiation on its own.

References

[1] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15:8:1–8:29, July 2012.

[2] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*, CSF 2012, pages 171–185. IEEE, 2012.

[3] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. USENIX Association.

[4] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[5] Dominique Devriese, Marco Patrignani, and Frank Piessens. Parametricity versus the universal type. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[6] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 371–384, New York, NY, USA, 2013. ACM.

[7] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. Local memory via layout randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 161–174, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[9] Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.

[10] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, July 2016. To appear.

[11] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP '13*, pages 10:1–10:1. ACM, 2013.

[12] James H. Morris, Jr. Protection in programming languages. *Commun. ACM*, 16:15–21, 1973.

[13] Joachim Parrow. General conditions for full abstraction. *Mathematical Structures in Computer Science*, 26(4):655–657, 2016.

[14] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.*, 37:6:1–6:50, April 2015.

[15] Marco Patrignani and Dave Clarke. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures*, 42(0):22 – 45, 2015.

[16] Marco Patrignani, Dominique Devriese, and Frank Piessens. On Modular and Fully Abstract Compilation. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium*, CSF 2016, 2016.

[17] ARM. ARMSecurity Technology. Building a secure system using trustzone technology. arm technical white paper, 2009.