Robustly-Safe Compilation –
Technical Appendix

Marco Patrignani\textsuperscript{1,2} Deepak Garg\textsuperscript{3}
\textsuperscript{1} Stanford University
\textsuperscript{2} CISPA Helmholtz Center for Information Security
\textsuperscript{3} Max Planck Institute for Software Systems

Contents

1 The Untyped Source Language: \( L^U \) \hfill 4
\hspace{1em} 1.1 Syntax \hfill 4
\hspace{1em} 1.2 Dynamic Semantics \hfill 4
\hspace{1em} 1.2.1 Component Semantics \hfill 5
\hspace{1em} 1.3 Monitor Semantics \hfill 7

2 The Target Language: \( L^P \) \hfill 8
\hspace{1em} 2.1 Syntax \hfill 8
\hspace{1em} 2.2 Operational Semantics of \( L^P \) \hfill 8
\hspace{1em} 2.2.1 Component Semantics \hfill 9
\hspace{1em} 2.3 Monitor Semantics \hfill 11

3 Language and Compiler Properties \hfill 12
\hspace{1em} 3.1 Safety, Attackers and Robust Safety \hfill 12
\hspace{1em} 3.2 Monitor Agreement and Attacker for \( L^P \) and \( L^U \) \hfill 12
\hspace{1em} 3.3 Cross-language Relations \hfill 13
\hspace{1em} 3.4 Correct and Robustly-safe Compilation \hfill 15
\hspace{1em} 3.4.1 Alternative definition for RSC \hfill 16
\hspace{1em} 3.4.2 Compiling Monitors \hfill 16

4 Compiler from \( L^U \) to \( L^P \) \hfill 17
\hspace{1em} 4.1 Properties of the \( \llbracket \bracket_{L^P}^{L^U} \) Compiler \hfill 18
\hspace{1em} 4.2 Back-translation from \( L^P \) to \( L^U \) \hfill 18
\hspace{1em} 4.2.1 Values Backtranslation \hfill 18
\hspace{1em} 4.2.2 Skeleton \hfill 19
\hspace{1em} 4.2.3 Single Action Translation \hfill 20
\hspace{1em} 4.2.4 The Back-translation Algorithm \( \langle\rangle_{L^P}^{L^U} \) \hfill 25
\hspace{1em} 4.2.5 Correctness of the Back-translation \hfill 25
\hspace{1em} 4.2.6 Remark on the Backtranslation \hfill 26

5 The Source Language: \( L^\tau \) \hfill 27
\hspace{1em} 5.1 Static Semantics of \( L^\tau \) \hfill 28
\hspace{1em} 5.1.1 Auxiliary Functions \hfill 28
\hspace{1em} 5.1.2 Typing Rules \hfill 28
\hspace{1em} 5.1.3 \textsf{UN} Typing \hfill 31
5.2 Dynamic Semantics of \( L^\tau \) .................................................. 32
5.2.1 Component Semantics .................................................. 33

6 \( L^\pi \): Extending \( L^\Pi \) with Concurrency and Informed Monitors 36
6.1 Syntax .............................................................. 36
6.2 Dynamic Semantics .................................................. 36
6.2.1 Component Semantics .................................................. 37

7 Extended Language Properties and Necessities 38
7.1 Monitor Agreement for \( L^\tau \) and \( L^\pi \) ......................... 38
7.2 Properties of \( L^\tau \) .................................................. 38
7.3 Properties of \( L^\pi \) .................................................. 39

8 Compiler from \( L^\tau \) to \( L^\pi \) ............................................. 40
8.1 Assumed Relation between \( L^\tau \) and \( L^\pi \) Elements .............. 40
8.2 Compiler Definition .................................................. 40
8.3 Properties of the \( L^\tau \)-\( L^\pi \) Compiler ........................................ 45
8.4 Cross-language Relation \( \equiv_\beta \) .................................................. 45

9 \( RSC \): Third Instance with Target Memory Isolation 47
9.1 \( L^I \), a Target Language with Memory Isolation ...................... 47
9.2 Compiler from \( L^\tau \) to \( L^I \) .................................................. 47

10 The Second Target Language: \( L^I \) .............................................. 48
10.1 Syntax .............................................................. 48
10.2 Operational Semantics of \( L^I \) .................................................. 49
10.2.1 Component Semantics .................................................. 49
10.3 Monitor Semantics .................................................. 52
10.4 Monitor Agreement for \( L^I \) .................................................. 52
10.5 Properties of \( L^I \) .................................................. 53

11 Second Compiler from \( L^\tau \) to \( L^I \) ............................................. 53
11.1 Properties of the \( L^\tau \)-\( L^I \) Compiler .................................................. 57
11.2 Cross-language Relation \( \equiv_\varphi \) .................................................. 57

12 Proofs .............................................................. 59
12.1 Proof of Theorem 1 (\( PF-RSC \) and \( RSC \) are equivalent) ............. 59
12.2 Proof of Theorem 2 (Compiler \( \parallel_L^U \) is \( CC \) ) ......................... 60
12.3 Proof of Theorem 3 (Compiler \( \parallel_L^U \) is \( RSC \) ) ......................... 64
12.4 Proof of Lemma 1 (Compiled code steps imply existence of source steps) .................................................. 65
12.5 Proof of Theorem 4 (\( \llangle \cdot \rrangle_{L^P} \) is correct) ......................... 67
12.6 Proof of Theorem 5 (Typability Implies Robust Safety in \( L^\tau \) ) ....... 72
12.6.1 Proof of Lemma 4 (Semantics and typed attackers coincide) ........... 72
12.7 Proof of Theorem 6 (Compiler \( \parallel_L^U \) is \( CC \) ) ......................... 76
12.8 Proof of Theorem 7 (Compiler \( \parallel_L^U \) is \( RSC \) ) ......................... 79
1 The Untyped Source Language: $L^U$

This is a sequential while language with monitors.

1.1 Syntax

**Whole Programs** $P ::= \ell_{\text{root}}; H; \overrightarrow{F}; I$

**Components** $C ::= \ell_{\text{root}}; \overrightarrow{F}; I$

**Contexts** $\Lambda ::= H; \overrightarrow{F} [\cdot]$

**Interfaces** $I ::= f$

**Functions** $F ::= f(x) \mapsto s; \text{return}$

**Operations** $\oplus ::= + | -$  

**Comparison** $\otimes ::= == | < | >$

**Values** $v ::= b \in \{\text{true}, \text{false}\} | n \in \mathbb{N} | \langle v, v \rangle | \ell$

**Expressions** $e ::= x | v | e \oplus e | e \otimes e | \langle e, e \rangle | e.1 | e.2 | !e$

**Statements** $s ::= \text{skip} | s; s | \text{let } x = e \text{ in } s | \text{if } e \text{ then } s \text{ else } s$

**Eval.Ctxs.** $E ::= [] | e \oplus E | E \oplus n | e \otimes E | E \otimes n$

$| \langle e, E \rangle | \langle E, v \rangle | E.1 | E.2 | !E$

**Heaps** $H ::= \emptyset \lor H; \ell \mapsto v$

**Monitors** $M ::= (\{\sigma\}, \leadsto, \sigma_0, \ell_{\text{root}}, \sigma_c)$

**Mon. States** $\sigma \in S$

**Mon. Reds.** $\leadsto ::= \emptyset \lor \leadsto; (s, H, s)$

**Substitutions** $\rho ::= \emptyset \lor [v / x]$

**Prog. States** $\Omega ::= C, H \triangleright (s)_{T}$

**Labels** $\lambda ::= \epsilon | \alpha$

**Actions** $\alpha ::= \text{call } f \triangleright H? \lor \text{call } f \triangleright H! \lor \text{ret } H! \lor \text{ret } H?$

**Traces** $\overrightarrow{\alpha} ::= \emptyset \lor \overrightarrow{\alpha} \cdot \alpha$

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.
1.2.1 Component Semantics

Expression \( e \) reduces to \( e' \).

Statement \( s \) reduces to \( s' \) and evolves the rest accordingly, emitting label \( \lambda \).

Program state \( \Omega \) steps to \( \Omega' \) emitting trace \( \pi \).
\[\begin{align*}
\text{(EL^{\text{u}-\text{false}})} & \\
H \triangleright e & \iff \text{false} \\
C, H \triangleright \text{if } e \text{ then } s \text{ else } s' & \rightarrow C, H \triangleright s \\
\text{(EL^{\text{u}-\text{letin}})} & \\
H \triangleright e & \rightarrow v \\
C, H \triangleright \text{let } x = e \text{ in } s & \rightarrow C, H \triangleright s[v / x] \\
\text{(EL^{\text{u}-\text{alloc}})} & \\
H \triangleright e & \rightarrow v \quad \ell \notin \text{dom}(H) \\
C, H \triangleright \text{new } e \text{ in } s & \rightarrow C, H; \ell \rightarrow v \triangleright s[\ell / x] \\
\text{(EL^{\text{u}-\text{update}})} & \\
H \triangleright e & \rightarrow v \\
H = H_3; \ell \rightarrow v'; H_2 & \quad H' = H_3; \ell \rightarrow v; H_2 \\
C, H \triangleright \ell := e & \rightarrow C, H' \triangleright \text{skip} \\
\text{(EL^{\text{u}-\text{call-internal}})} & \\
C, \text{infs} \vdash f, f': \text{internal} & \quad F' = F'; f' \\
f(x) \rightarrow s; \text{return}; \in C.\text{funs} & \quad H \triangleright e \rightarrow v \\
C, H \triangleright \text{(call } f e)_{\ell} & \rightarrow C, H \triangleright (s; \text{return};[v / x])_{F', f} \\
\text{(EL^{\text{u}-\text{callback}})} & \\
\text{F} = \text{F'; f} & \quad f(x) \rightarrow s; \text{return}; \in C.\text{funs} \\
C, \text{infs} \vdash f', f: \text{out} & \quad H \triangleright e \rightarrow v \\
C, H \triangleright \text{(call } f e)_{\ell} & \rightarrow C, H \triangleright (s; \text{return};[v / x])_{F', f} \\
\text{(EL^{\text{u}-\text{call}})} & \\
\text{F} = \text{F'; f} & \quad f(x) \rightarrow s; \text{return}; \in C.\text{funs} \\
C, \text{infs} \vdash f', f: \text{in} & \quad H \triangleright e \rightarrow v \\
C, H \triangleright \text{(call } f e)_{\ell} & \rightarrow C, H \triangleright (s; \text{return};[v / x])_{F', f} \\
\text{(EL^{\text{u}-\text{ret-internal}})} & \\
\text{F} = \text{F'; f} & \quad C, \text{infs} \vdash f, f': \text{internal} \\
C, H \triangleright \text{(return);}_{F', f} & \rightarrow C, H \triangleright \text{(skip)}_{\ell} \\
\text{(EL^{\text{u}-\text{retback}})} & \\
\text{F} = \text{F'; f} & \quad C, \text{infs} \vdash f, f': \text{in} \\
C, H \triangleright \text{(return);}_{F', f} & \rightarrow H \triangleright C, H \triangleright \text{(skip)}_{\ell} \\
\text{(EL^{\text{u}-\text{return}})} & \\
\text{F} = \text{F'; f} & \quad C, \text{infs} \vdash f, f': \text{out} \\
C, H \triangleright \text{(return);}_{F', f} & \rightarrow H \triangleright C, H \triangleright \text{(skip)}_{\ell} \\
\end{align*}\]
### 1.3 Monitor Semantics

Let $\text{reach}(\ell_0, H)$ return a set of locations $\{\ell\}$ in $H$ such that it is possible to reach any $\ell \in \{\ell\}$ from $\ell_0$ just by expression evaluation.

$$\text{reach}(\ell, H) = \{\ell \mid \exists e. H \triangleright e \Rightarrow \ell \land \ell \in \text{dom}(H)\}$$

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

\[
\begin{align*}
M; H & \rightsquigarrow M' \\
(M; H \rightsquigarrow M') & = (\{\sigma\}, \rightsquigarrow, \sigma_0, \ell_{\text{root}}, \sigma_f) \\
& \text{(L\textsubscript{UL}-Monitor Step)} \\
(\sigma_c, H', \sigma_f) & \in \rightsquigarrow H' \subseteq H \\
& \text{dom}(H') = \text{reach}(\ell_{\text{root}}, H) \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigarrow M' \\
M; H & \rightsquigaro
\end{align*}
\]

Monitor actions are the only part of traces that matter for safety, so we define function $\text{relevant}(\cdot)$ 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.

\[
\begin{align*}
\text{relevant}(\emptyset) & = \emptyset \\
\text{relevant}(\text{call } f v H? \cdot \overline{\alpha}) & = H \cdot \text{relevant}(\overline{\pi}) \\
\text{relevant}(\text{call } f v H! \cdot \overline{\alpha}) & = H \cdot \text{relevant}(\overline{\pi}) \\
\text{relevant}(\text{ret } H! \cdot \overline{\alpha}) & = H \cdot \text{relevant}(\overline{\pi}) \\
\text{relevant}(\text{ret } H? \cdot \overline{\alpha}) & = H \cdot \text{relevant}(\overline{\pi})
\end{align*}
\]
2 The Target Language: \textbf{L}^\text{P}

2.1 Syntax

\begin{itemize}
  \item \textbf{Whole Programs} \( P ::= k_{\text{root}} : F : I \)
  \item \textbf{Components} \( C ::= k_{\text{root}} : F : I \)
  \item \textbf{Contexts} \( A ::= F [ \cdot ] \)
  \item \textbf{Interfaces} \( I ::= f \)
  \item \textbf{Functions} \( F ::= f(x) \mapsto \text{return}; \)
  \item \textbf{Operations} \( \oplus ::= + | - \)
  \item \textbf{Comparison} \( \otimes ::= == | < | > \)
  \item \textbf{Values} \( v ::= n \in \mathbb{N} | (v, v) | k \)
  \item \textbf{Expressions} \( e ::= x | v | e \oplus e | e \otimes e | \langle e, e \rangle | e.1 | e.2 | e \text{ with } e \)
  \item \textbf{Statements} \( s ::= \text{skip} | s; s | \text{let } x = e \text{ in } s | \text{ifz } e \text{ then } s \text{ else } s | \text{call } f e \)
    \begin{itemize}
      \item \( x ::= e \text{ with } e \)
      \item \( \text{let } x = \text{new } e \text{ in } s \)
      \item \( \text{let } x = \text{hide } e \text{ in } s \)
    \end{itemize}
  \item \textbf{Eval. Cts.} \( E ::= [ \cdot ] | e \oplus E | E \oplus n | e \otimes E | E \otimes n | !E \text{ with } v | !e \text{ with } E \)
    \begin{itemize}
      \item \( \langle e, E \rangle | \langle E, v \rangle | E.1 | E.2 \)
    \end{itemize}
  \item \textbf{Heaps} \( H ::= \emptyset | H ; n \mapsto v : \eta | H ; k \)
  \item \textbf{Tag} \( \eta ::= \bot | k \)
  \item \textbf{Monitors} \( M ::= \{ \sigma \}, \rightarrow, \sigma_0, k_{\text{root}}, \sigma_c \)
  \item \textbf{Mon. States} \( \sigma \in S \)
  \item \textbf{Mon. Redns.} \( \rightsquigarrow ::= \emptyset | \rightsquigarrow (s, H, s) \)
  \item \textbf{Substitutions} \( \rho ::= \emptyset | \rho[v / x] \)
  \item \textbf{Prog. States} \( \Omega ::= C, H \triangleright (s)_{\Gamma} \)
  \item \textbf{Labels} \( \lambda ::= \epsilon | \alpha \)
  \item \textbf{Actions} \( \alpha ::= \text{call } f v H? | \text{call } f v H! | \text{return } H! | \text{return } H? \)
  \item \textbf{Traces} \( \pi ::= \emptyset | \pi \cdot \alpha \)
\end{itemize}

2.2 Operational Semantics of \textbf{L}^\text{P}

\begin{tabular}{c c c c}
  \hline
  \textbf{Helpers} & \hline
  \( (\text{L}^\text{P}-\text{Jump-Internal}) \) & \( (\text{L}^\text{P}-\text{Jump-IN}) \) & \( (\text{L}^\text{P}-\text{Jump-OUT}) \)\\
  \hline
  \( (f' \in I \wedge f \in I) \lor (f' \notin I \wedge f \notin I) \) & \( f \in I \wedge f' \notin I \) & \( f \notin I \wedge f' \in I \)\\
  \( I \vdash f, f' : \text{internal} \) & \( I \vdash f : \text{in} \) & \( I \vdash f : \text{out} \)\\
  \hline
\end{tabular}
2.2.1 Component Semantics

\[ H \triangleright e \quad \iff \quad e' \]

Expression \( e \) reduces to \( e' \).

\[ C, H \triangleright s \quad \rightarrow \quad C', H' \triangleright s' \]

Statement \( s \) reduces to \( s' \) and evolves the rest accordingly, emitting label \( \lambda \).

\[ \Omega \xrightarrow{\pi} \Omega' \]

Program state \( \Omega \) steps to \( \Omega' \) emitting trace \( \pi \).

\[ C, H \triangleright s \quad \rightarrow \quad C'; H' \triangleright s' \]

\[ C, H \triangleright \text{skip}; s \quad \rightarrow \quad C, H \triangleright s \]

\[ C, H \triangleright \text{ifz e then s else s'} \quad \rightarrow \quad C, H \triangleright s \]
2.3 Monitor Semantics

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

\[
\text{reach}(n_r, k_r, H) = \left\{ n \mid H \triangleright e \leftrightarrow !n \text{ with } v \leftrightarrow v' \quad \text{and} \quad \text{fv}(e) = n_r \cup k_r \right\}
\]

\[
M; H \rightsquigarrow M'
\]

\[
\begin{align*}
\text{(LP-Monitor Step)} \\
M = (\{\sigma\}, \rightsquigarrow, \sigma_0, k_{\text{root}}, \sigma_c) & \quad M' = (\{\sigma\}, \rightsquigarrow, \sigma_0, k_{\text{root}}, \sigma_f) \\
(s_c, H', s_r) \in \rightsquigarrow H' \subseteq H & \quad \text{dom}(H') = \text{reach}(0, k_{\text{root}}, H)
\end{align*}
\]

\[
\begin{align*}
\text{(LP-Monitor Step Trace Base)} \\
M; \emptyset \rightsquigarrow M & \quad M; H \rightsquigarrow M' \\
\text{(LP-Monitor Step Trace)} & \quad M; H' \rightsquigarrow M' \\
\text{(LP-safe trace)} & \quad M; H \rightsquigarrow M' \\
\text{relevant}(\pi) = H & \quad M \vdash \pi
\end{align*}
\]
3 Language and Compiler Properties

3.1 Safety, Attackers and Robust Safety

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

**Definition 1** (Safety).

\[ M \vdash C : \text{safe} \overset{\text{def}}{=} \]

\[ \text{if } \vdash C : \text{whole} \]

\[ \text{then } \text{if } \Omega_0(C) \xrightarrow{\alpha} _{\_} \]

\[ \text{then } M \vdash \alpha \]

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

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

\[ C \vdash A : \text{attacker} \overset{\text{def}}{=} \text{no location the component cares about } \in \text{fn}(A) \]

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).

\[ M \vdash C : \text{rs} \overset{\text{def}}{=} \forall A. \]

\[ \text{if } M \sim C \]

\[ C \vdash A : \text{attacker} \]

\[ \text{then } M \vdash A [C] : \text{safe} \]

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

The definition of \( M \sim C \) is to be specified on a language-specific basis, as the next section does for \( L^U \) and \( L^P \).

3.2 Monitor Agreement and Attacker for \( L^P \) and \( L^U \)

**Definition 4** (\( L^U \): \( M \sim C \)).

\[ (\{\sigma\}, \sim, \sigma_0, \ell_{\text{root}}, \sigma_c) \sim (\ell_{\text{root}}; \overline{F}; \overline{I}) \]

A monitor and a component agree if they focus on the same initial location \( \ell_{\text{root}} \).

**Definition 5** (\( L^P \): \( M \sim C \)).

\[ (\{\sigma\}, \sim, \sigma_0, k_{\text{root}}, \sigma_c) \sim (k_{\text{root}}; \overline{F}; \overline{I}) \]

12
A monitor and a component agree if they use the same capability $k_{\text{root}}$ to protect the initial location $0$.

**Definition 6** ($L^U$ attacker).

$$C \vdash A : \text{attacker} \equiv C = (\ell_{\text{root}}; F; I), A = H; F'$$

$$\ell_{\text{root}} \notin (\text{fn}(F') \cup H)$$

**Definition 7** ($L^P$ attacker).

$$C \vdash A : \text{attacker} \equiv C = (k_{\text{root}}; F; I), A = F'$$

$$k_{\text{root}} \notin \text{fn}(F')$$

### 3.3 Cross-language Relations

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

- if $(\ell_1, n, \eta) \in \beta$ and $(\ell_2, n, \eta)$ then $\ell_1 = \ell_2$;
- if $(\ell, n_1, \eta_1) \in \beta$ and $(\ell, n_2, \eta_2)$ then $n_1 = n_2$ 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 $\beta_0$ which relates initial locations of monitors.

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

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

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

- $\text{true} \approx_{\beta} 0$;
- $\text{false} \approx_{\beta} n$ where $n \neq 0$;
- $\ell \approx_{\beta} \langle n, v \rangle$ where
  $$\begin{cases} 
  v = k & \text{if } (\ell, n, k) \in \beta \\
  v \neq k & \text{otherwise, so } (\ell, n, \bot) \in \beta
  \end{cases}$$
- $\langle v_1, v_2 \rangle \approx_{\beta} \langle v_1, v_2 \rangle$ iff $v_1 \approx_{\beta} v_1$ and $v_2 \approx_{\beta} v_2$.

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 $\beta$ is not needed as states do not have locations inside.

Function names are related when they are the same: $f \approx_{\beta} f$. 13
Variables names are related when they are the same: $x \approx_\beta x$.

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

\[ \alpha \approx_\beta \alpha \]

\[ f \approx f \]
\[ v \approx_\beta v \]
\[ H \approx_\beta H \]

\[ \text{(Call relation)} \]

\[ \text{call } f \text{? } \approx_\beta \text{ call } f \text{? } \]
\[ H \approx_\beta H \]

\[ \text{(Callback relation)} \]

\[ \text{ret } H! \approx_\beta \text{ ret } H! \]

\[ \text{(Return relation)} \]

\[ H \approx_\beta ret \]

\[ \epsilon \approx_\beta \epsilon \]

\[ H \approx_\beta H \]

\[ \epsilon \approx_\beta \epsilon \]

\[ \text{(Heap relation)} \]

\[ H \approx_\beta H \]

\[ H \approx_\beta H_1; H_2 \]
\[ \ell \approx_\beta (\langle n, \_ \rangle) \]
\[ v \approx_\beta v \]
\[ H = H_1; n \mapsto v : \eta; H_2 \]
\[ H; \ell \mapsto v \approx_\beta H \]

\[ \emptyset \approx_\beta \emptyset \]

\[ \text{(Empty relation)} \]

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

\[ \text{As additional notation for states, we define when a state is stuck as follows} \]

\[ \Omega = M; T; H \triangleright s \quad s \neq \text{skip} \quad \tilde{\#} \Omega', \lambda. \Omega \xrightarrow{\lambda} \Omega' \]

\[ \Omega^\times \]
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):

\[
\begin{align*}
\Omega &= M; \overline{F}; \overline{T}; H \triangleright \text{skip} \\
\Omega^\triangleright
\end{align*}
\]

\[
\begin{align*}
\Omega &= M; \overline{F}; \overline{T}; H \triangleright \Pi \\
\forall \pi \in \Pi. M; \overline{F}; \overline{T}; H \triangleright \pi^\triangleright \\
\Omega^\triangleright
\end{align*}
\]

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).

\[
\Omega \approx_\beta \Omega
\]

(Related states - Whole)

\[
\begin{align*}
\Omega &= M; \overline{F}; \overline{T}; H \triangleright s \\
M \approx_\beta M & \quad \Pi \approx_\beta \Pi \\
\Omega \approx_\beta \Omega
\end{align*}
\]

### 3.4 Correct and Robustly-safe Compilation

Consider a compiler to be a function of this form: \( \llbracket \cdot \rrbracket^S : \mathbb{C} \rightarrow \mathbb{C} \), taking a source component and producing a target component.

**Definition 10** (Correct Compilation).

\[
\llbracket \cdot \rrbracket^S_T : \mathbb{C} \xrightarrow{\text{def}} \forall \mathbb{C}, \exists \beta.
\]

if

\[
\begin{align*}
\Omega_0 \left( \llbracket \cdot \rrbracket^S_T \right) \xrightarrow{\pi} \Omega \\
\Omega^\triangleright \\
\Omega_0 \left( \mathcal{C} \right) \approx_\beta \Omega_0 \left( \llbracket \cdot \rrbracket^S_T \right)
\end{align*}
\]

then

\[
\begin{align*}
\Omega_0 \left( \mathcal{C} \right) & \xrightarrow{\pi} \Omega \\
\beta_0 \subseteq \beta \\
\Omega \approx_\beta \Omega \\
\overline{\alpha} \approx_\beta \overline{\alpha} \\
\Omega^\triangleright
\end{align*}
\]

Technically, any sequence \( \overline{\alpha} \) above is empty, as \( \overline{I} \) is empty (the program is whole).
**Definition 11** (Robust Safety Preserving Compilation).

\[
\vdash \llbracket T \rrbracket_T : RSC \overset{\text{def}}{=} \forall C, M, M. \\
\text{if } M \vdash C : rs \\
M \approx M \\
\text{then } M \vdash \llbracket C \rrbracket_T^S : rs
\]

3.4.1 Alternative definition for RSC

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

\[
\vdash \llbracket T \rrbracket_T : PF-RSC \overset{\text{def}}{=} \forall C. \\
\text{if } \forall A, \pi. \\
[\llbracket C \rrbracket_T]_T \vdash A : attacker \\
\vdash A \llbracket C \rrbracket_T : whole \\
\Omega_0 ([\llbracket C \rrbracket_T^S]_T) \xrightarrow{\pi} _\_ \\
\text{then } \exists A, \pi. \\
C \vdash A : attacker \\
\vdash A \llbracket C \rrbracket : whole \\
\Omega_0 (C) \xrightarrow{\pi} _\_ \\
\text{relevant}(\pi) \approx \beta \text{ relevant}(\pi)
\]

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

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

\[
\forall \llbracket T \rrbracket_T, \vdash \llbracket T \rrbracket_T : PF-RSC \iff \vdash \llbracket T \rrbracket_T : RSC
\]

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:

\[
\llbracket T \rrbracket_T^S : C \rightarrow C.
\]

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

\[
\vdash \llbracket T \rrbracket_T^S : rs-pres(M) \overset{\text{def}}{=} \forall C, M. \\
\text{if } M \vdash C : rs \\
\text{then } \llbracket M \rrbracket_T^S \vdash \llbracket C \rrbracket_T^S : rs
\]
4 Compiler from $\mathbf{L}^U$ to $\mathbf{L}^P$

**Definition 14 (Compiler $\mathbf{L}^U$ to $\mathbf{L}^P$).** $\llbracket \cdot \rrbracket_{\mathbf{L}^P} : \mathcal{C} \rightarrow \mathcal{C}$

$\llbracket \cdot \rrbracket_{\mathbf{L}^P}$ is defined as follows:

$$
\llbracket \ell_{\text{root}} : F; I, M \rrbracket_{\mathbf{L}^P} = k_{\text{root}} : \llbracket F \rrbracket_{\mathbf{L}^P} : \llbracket I \rrbracket_{\mathbf{L}^P} : \llbracket \cdot \rrbracket_{\mathbf{L}^P}$$

if $\ell_{\text{root}} \approx_\beta \langle 0, k_{\text{root}} \rangle$

$$
\llbracket f(x) \rightarrow s; \text{return} \rrbracket_{\mathbf{L}^P} = f(x) \rightarrow \llbracket s \rrbracket_{\mathbf{L}^P}; \text{return;}
$$

$$
\llbracket f \rrbracket_{\mathbf{L}^P} = f
$$

**Expressions**

- $\llbracket \text{true} \rrbracket_{\mathbf{L}^P} = 0$ if $\text{true} \approx_\beta 0$

- $\llbracket \text{false} \rrbracket_{\mathbf{L}^P} = 1$ if $\text{false} \approx_\beta 1$

- $\llbracket n \rrbracket_{\mathbf{L}^P} = n$ if $n \approx_\beta n$

- $\llbracket x \rrbracket_{\mathbf{L}^P} = x$

- $\llbracket \ell \rrbracket_{\mathbf{L}^P} = \langle n, v \rangle$ if $\ell \approx_\beta \langle n, v \rangle$

- $\llbracket (e_1, e_2) \rrbracket_{\mathbf{L}^P} = \langle \llbracket e_1 \rrbracket_{\mathbf{L}^P}, \llbracket e_2 \rrbracket_{\mathbf{L}^P} \rangle$

- $\llbracket e.1 \rrbracket_{\mathbf{L}^P} = \llbracket e \rrbracket_{\mathbf{L}^P}.1$

- $\llbracket e.2 \rrbracket_{\mathbf{L}^P} = \llbracket e \rrbracket_{\mathbf{L}^P}.2$

- $\llbracket e + e' \rrbracket_{\mathbf{L}^P} = \llbracket e \rrbracket_{\mathbf{L}^P} + \llbracket e' \rrbracket_{\mathbf{L}^P}$

- $\llbracket e \odot e' \rrbracket_{\mathbf{L}^P} = \llbracket e \rrbracket_{\mathbf{L}^P} \odot \llbracket e' \rrbracket_{\mathbf{L}^P}$

**Statements**

- $\llbracket \text{skip} \rrbracket_{\mathbf{L}^P} = \text{skip}$

- $\llbracket s_1 \odot s_2 \rrbracket_{\mathbf{L}^P} = \llbracket s_1 \rrbracket_{\mathbf{L}^P} \odot \llbracket s_2 \rrbracket_{\mathbf{L}^P}$

- $\llbracket \text{let } x = e \text{ in } s \rrbracket_{\mathbf{L}^P} = \text{let } x = \llbracket e \rrbracket_{\mathbf{L}^P} \text{ in } \llbracket s \rrbracket_{\mathbf{L}^P}$

- $\llbracket \text{if } e \text{ then } s_1 \text{ else } s_2 \rrbracket_{\mathbf{L}^P} = \text{ifz } \llbracket e \rrbracket_{\mathbf{L}^P} \text{ then } \llbracket s_1 \rrbracket_{\mathbf{L}^P} \text{ else } \llbracket s_2 \rrbracket_{\mathbf{L}^P}$

- $\llbracket \text{let } x \text{ new } e \text{ in } s \rrbracket_{\mathbf{L}^P} = \text{let } x_{\text{loc}} = \text{new } \llbracket e \rrbracket_{\mathbf{L}^P} \text{ in }$

  - let $x_{\text{cap}} = \text{hide } x_{\text{loc}}$ in

  - let $x = \langle x_{\text{loc}}, x_{\text{cap}} \rangle$ in $\llbracket s \rrbracket_{\mathbf{L}^P}$
\[ x \leftarrow e \]_L^U = \begin{aligned} \text{let } x_1 &= x.1 \text{ in} \\ &\text{let } x_2 = x.2 \text{ in} \\ x_1 &= \lfloor e \rfloor_{L^P} \text{ with } x_2 \end{aligned} \quad (\lbrack \cdot \rfloor_{L^P}^{L^U} - \text{Assign})

\[ \text{call f e} \]_L^U = \begin{aligned} \text{call f } \lfloor e \rfloor_{L^P} \end{aligned} \quad (\lbrack \cdot \rfloor_{L^P}^{L^U} - \text{call})

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

\[ \[ v / x \rfloor_{L^P}^{L^U} = \lfloor v \rfloor_{L^P}^{L^U} / x \]

**Optimisation** We could optimise Rule (\lbrack \cdot \rfloor_{L^P}^{L^U} - \text{Deref}) as follows:

- rename the current expressions except dereferencing to \( b \);

- reform expressions both in \( L^P \) and \( L^U \) as \( e := b | \text{let } x = b \text{ in } e | !b \). In the case of \( L^P \) it would be \( \cdots | !b \text{ with } b \).

This allows expressions to compute e.g., pairs and projections.

- rewrite the Rule (\lbrack \cdot \rfloor_{L^P}^{L^U} - \text{Deref}) case for compiling \( !b \) into:

\[ \text{let } x = \lfloor b \rfloor_{L^P} \text{ in let } x_1 = x.1 \text{ in let } x_2 = x.2 \text{ in } !x_1 \text{ with } x_2. \]

- 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 \( []_{L^P}^{L^U} \) Compiler

**Theorem 2** (Compiler \( []_{L^P}^{L^U} \) is \textit{CC}). \( \vdash []_{L^P}^{L^U} : CC \)

**Theorem 3** (Compiler \( []_{L^P}^{L^U} \) is \textit{RSC}). \( \vdash []_{L^P}^{L^U} : RSC \)

### 4.2 Back-translation from \( L^P \) to \( L^U \)

#### 4.2.1 Values Backtranslation

Here is how values are back translated.

\[
\langle \langle \cdot \rangle \rangle_{L^P}^{L^U} : \nu \rightarrow \nu
\]

\[
\langle \langle 0 \rangle \rangle_{L^P}^{L^U} = \text{true}
\]
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 \texttt{call f v H?} could result in either \texttt{call f true \textbar H?} or \texttt{call f 0 \textbar H?}. Now depending on \( f \)'s body, which is the component to be compiled, supplying \texttt{true} or \texttt{0} may have different outcomes. Let us assume that the compilation of \( f \), when receiving \texttt{call f v H?} does not get stuck. If \( f \) contains \texttt{if x then s else s'}, supplying \texttt{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).

\[
\forall \Omega'' \approx \beta \Omega''
\]

if \( \Omega'' \approx \beta \Omega'' \)

\[
\Omega'' \xrightarrow{\alpha?} C, H \triangleright [s]_{L^P}^U; s'\rho
\]

\[
C, H \triangleright [s]_{L^P}^U; s'\rho \xrightarrow{\alpha!} \Omega'
\]

\[
\{\alpha?\} = \{\alpha? \mid \alpha? \approx \beta \alpha?\}
\]

\[
\{\rho\} = \{\rho \mid \rho \approx \beta \rho\}
\]

then \( \exists \alpha_j? \in \{\alpha?\}, \rho_y \in \{\rho\} ; C_j, H_j \triangleright s_j; s'_j\rho_j'. \)

if \( \Omega'' \xrightarrow{\alpha?} C_j, H_j \triangleright s_j; s'_j\rho_j' \)

then \( C_j, H_j \triangleright s_j; s'_j\rho_j \approx \beta C, H \triangleright [s]_{L^P}^U; s'\rho \)

\[
C_j, H_j \triangleright s_j; s'_j\rho_j \xrightarrow{\alpha!} \Omega'
\]

\[
\alpha! \approx \beta \alpha!
\]

\[
\Omega' \approx \beta \Omega'
\]

### 4.2.2 Skeleton

\[
\langle \circ \circ \rangle_{L^P}^U : I \rightarrow F
\]

\[
\langle \circ \circ \rangle_{L^P}^U = f(x) \rightarrow \text{incrementCounter(); return;}
\]

\[
\langle \circ \circ \rangle_{L^P}^U - \text{fun}
\]
Functions call \texttt{incrementCounter()} before returning to ensure that when a return back is modelled, the counter is incremented right before returning and not beforehand, as doing so would cause the possible execution of other backward-translated code blocks. Its implementation is described below.

\[
\langle\langle \cdot \rangle\rangle_{L_U} : I \to A
\]

\[
\langle\langle I \rangle\rangle_{L_U}^{L_P} = \ell_i \mapsto 1; \\
\ell_{\text{glob}} \mapsto 0 \\
\text{main}(x) \mapsto \text{incrementCounter}(); \text{return;} \\
\text{incrementCounter}() \mapsto \text{see below} \\
\text{register}(x) \mapsto \text{see below} \\
\text{update}(x) \mapsto \text{see below}
\]

\[
\langle\langle f \rangle\rangle_{L_U}^{L_P} \quad \forall f \in I
\]

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

The skeleton sets up the infrastructure. It allocates global locations \(\ell_i\), which is used as a counter to count steps in actions, and \(\ell_{\text{glob}}\), which is used to keep track of attacker knowledge, as described below. Then it creates a dummy for all functions expected in the interfaces \(I\) as well as a dummy for the \texttt{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 \texttt{ak} to indicate a list of pairs of locations and tag to access them \(\langle m, \eta \rangle\) that is what the context has access to. We use functions \texttt{.loc} to access obtain all locations of such a list and \texttt{.cap} to obtain all the capabilities (or 0 when \(\eta = \perp\)) of the list.

We use function \texttt{incrementCounter} to increment the contents of \(\ell_i\) by one.

\[
\text{incrementCounter}(\ ) \mapsto \\
\text{let } c = !\ell_i \text{ in let } l = \ell_i \text{ in } l := c + 1
\]

Starting from location \(\ell_g\) we keep a list whose elements are pairs locations-numbers, we indicate this list as \(L_{\text{glob}}\).

We use function \texttt{register(\(\ell, n\))} which adds the pair \(\langle \ell, n \rangle\) to the list \(L_{\text{glob}}\). Any time we use this we are sure we are adding a pair for which no other existing pair in \(L_{\text{glob}}\) has a second projection equal to \(n\). This function can be
defined as follows:
\[
\text{register}(x) \mapsto \\
\text{let } x_1 = x.1 \\
in \text{let } x_n = x.2 \\
in \text{L}_{\text{glob}} :: (x_1, x_n)
\]

$L_{\text{glob}}$ 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 0. Where :: is a recursive function that starts from $\ell_{\text{glob}}$ and looks for its last element (i.e., it performs second projections until it hits a 0), then replaces that second projection with $\langle\langle x_1, x_n \rangle, 0 \rangle$.

**Lemma 2** ($\text{register}(\ell, n)$ does not add duplicates for $n$). For $n$ supplied as parameter by $\llbracket L_{\text{P}}^P, C; H \triangleright \text{register}(\ell, n) \rightarrow^L C; H' \triangleright \text{skip} \land \langle_, n \rangle \notin L_{\text{glob}}$.

**Proof.** Simple analysis of Rules ($\llbracket L_{\text{P}}^P, C; H \triangleright \text{L}_{\text{loc}}$) to ($\llbracket L_{\text{P}}^P, C; H' \triangleright \text{L}_{\text{loc}}$). □

We use function $\text{update}(n, v)$ which accesses the elements in the $L_{\text{glob}}$ 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** ($\text{update}(n, v)$ never gets stuck). $C; H \triangleright \text{update}(n, v) \rightarrow^L C; H' \triangleright \text{skip}$ for $n$ and $v$ supplied as parameters by $\llbracket L_{\text{P}}^P \land H' = H[\ell \mapsto v / \ell \mapsto \_]$ for $\ell \approx_\beta (n, \_)$.

**Proof.** Simple analysis of Rules ($\llbracket L_{\text{P}}^P, C; H \triangleright \text{L}_{\text{loc}}$) and ($\llbracket L_{\text{P}}^P, C; H' \triangleright \text{L}_{\text{loc}}$). □

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

**Definition 15** (Reachable).

\[
\text{reachable}(H, v, ak) = \left\{ (n \mapsto v : k, e) \mid n \in \text{reach}(n_{st}, k_{st}, H) \right. \\
\text{where } n_{st} \in v \cup ak.loc \\
\text{and } k_{st} \in k_{root} \cup ak.cap \\
\text{and } n \mapsto v : k \in H \\
\text{and } H\triangleright e \iff \text{!n with } k \\
\text{and } \forall H. H \approx_\beta H \\
\text{and } (\ell, n, k) \in \beta \right. \right. \right. \}
\]
Intuitively, \( \text{reachable}(\cdot) \) 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, \( n \mapsto v : \eta \)). Additionally, it tells how to reach those locations in the source so that we can \( \text{register}(\cdot) \) them for the source attacker (the backtranslated context) to access.

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

\[
\begin{align*}
\langle \langle \cdot \rangle \rangle_{LU} : e & \to e \\
\langle \langle e \rangle \rangle_{LU}^L &= \langle \langle e \rangle \rangle_{LU}^P \\
\langle \langle e.1 \rangle \rangle_{LU}^L &= \langle \langle e \rangle \rangle_{LU}^P .1 \\
\langle \langle e.2 \rangle \rangle_{LU}^L &= \langle \langle e \rangle \rangle_{LU}^P .2
\end{align*}
\]

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.

\[
\begin{align*}
\langle \langle \cdot \rangle \rangle_{LU}^P : \alpha \times n & \in \mathbb{N} \times H \times \bar{n} \times \bar{\eta} \times \bar{f} \to \{ s \times \bar{n} \times \bar{\eta} \times H \times \bar{f} \times f \}
\end{align*}
\]

\[
\begin{align*}
\langle \langle \text{call } f \ v \ H^?, n, \ H_{\text{pre}}, \ ak, \bar{f} \rangle \rangle_{LU}^L & = \\
& \begin{cases} 
\text{if } ! \ell_i = n \text{ then} \\
n & \text{incrementCounter()} \\
& \text{let } x_1 = \text{new } v_1 \text{ in register}(\langle x_1, n_1 \rangle) \\
& \cdots \\
& \text{let } x_j = \text{new } v_j \text{ in register}(\langle x_j, n_j \rangle) \\
& \text{update}(m_1, u_1) \\
& \cdots \\
& \text{update}(m_i, u_i) \\
& \text{call } f \ v \\
& \text{else skip}
\end{cases}
\end{align*}
\]

where \( H \setminus H_{\text{pre}} = H_n \)

\[
\begin{align*}
& \quad H_n = n_1 \mapsto v_1 : \eta_1, \cdots, n_j \mapsto v_j : \eta_j \\
& \text{and } H \cap H_{\text{pre}} = H_c
\end{align*}
\]

\[
\begin{align*}
& \quad H_c = m_1 \mapsto u_1 : \eta'_1, \cdots, m_l \mapsto u_1 : \eta'_l \\
& \text{and } ak' = ak, (n_1, \eta_1), \cdots, (n_j, \eta_j)
\end{align*}
\]
\[
\text{and } \bar{f} = f^\dagger P
\]

\[
\begin{align*}
\left\langle \text{call } f \text{ v } H, n, H_{\text{pre}}, ak, \bar{f} \right\rangle_{L^P}^{L^P} &= \left\{ \begin{array}{l}
\text{if } !\ell_i = n \text{ then } \\
\text{incrementCounter}() \\
\text{let } l_1 = e_1 \text{ in register}((l_1, n_1)) \\
\vdots \\
\text{let } l_j = e_j \text{ in register}((l_j, n_j)) \\
\text{else skip} \\
\end{array} \right. \\
&\quad , ak', H', f, \bar{f}
\end{align*}
\]

(\langle \rangle_{L^P}^{L^P} - \text{callback-loc})

if \text{reachable}(H, v, ak) = \langle n_1 \mapsto v_1 : \eta_1, \ldots, n_j \mapsto v_j : \eta_j, e_j \rangle

and

\[
\begin{align*}
\left\langle \text{ret } H? \text{ v } H, n, H_{\text{pre}}, ak, f; \bar{f} \right\rangle_{L^P}^{L_P} &= \left\{ \begin{array}{l}
\text{if } !\ell_i = n \text{ then } \\
// no \text{ incrementCounter() as explained} \\
\text{let } x_1 = \text{new } v_1 \text{ in register}((x_1, n_1)) \\
\vdots \\
\text{let } x_j = \text{new } v_j \text{ in register}((x_j, n_j)) \\
\text{update}(m_1, u_1) \\
\vdots \\
\text{update}(m_j, u_j) \\
\text{else skip} \\
\end{array} \right. \\
&\quad , ak', H, f, \bar{f}
\end{align*}
\]

(\langle \rangle_{L^P}^{L^P} - \text{retback})

where

\[
H \setminus H_{\text{pre}} = H_n
\]

\[
H_n = n_1 \mapsto v_1 : \eta_1, \ldots, n_j \mapsto v_j : \eta_j
\]

and

\[
H \cap H_{\text{pre}} = H_c
\]

\[
H_c = m_1 \mapsto u_1 : \eta'_1, \ldots, m_j \mapsto u_j : \eta'_j
\]

and

\[
ak' = ak, (n_1, \eta_1), \ldots, (n_j, \eta_j)
\]

\[
\begin{align*}
\left\langle \text{ret } H! \text{ v } H, n, H_{\text{pre}}, ak, f; \bar{f} \right\rangle_{L^P}^{L^P} &= \left\{ \begin{array}{l}
\text{if } !\ell_i = n \text{ then } \\
\text{incrementCounter}() \\
\text{let } l_1 = e_1 \text{ in register}((l_1, n_1)) \\
\vdots \\
\text{let } l_j = e_j \text{ in register}((l_j, n_j)) \\
\text{else skip} \\
\end{array} \right. \\
&\quad , ak', H', f, \bar{f}'
\end{align*}
\]

(\langle \rangle_{L^P}^{L^P} - \text{ret-loc})

23
if \( \text{reachable}(H, 0, ak) = \langle n_1 \mapsto v_1 : e_1 \rangle, \cdots, \langle n_j \mapsto v_j : e_j \rangle \) and \( ak' = ak, (n_1, \eta_1), \cdots, (n_j, \eta_j) \) and \( \bar{f} = f'\bar{f} \)

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 \( \ell_i \) 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 \((H_n)\), allocating them and registering them in the \( L_{\text{glob}} \) list via the \( \text{register}(\cdot) \) function. These locations are also added to the attacker knowledge \( ak' \). Then we need to update the heap locations we already know of. These locations are \( H_c \) and as we know them already, we use the \( \text{update}(\cdot) \) 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 \( \text{reachable}(\cdot) \) function to return a list of pairs of locations \( n \) and expressions \( e \). We use \( n \) to expand the attacker knowledge \( ak' \) as these locations are now reachable. We use \( e \) to reach these locations in the source heap so that we can \( \text{register} \) them and ensure they are accessible through \( L_{\text{glob}} \).

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

\[
\begin{align*}
\langle \circ \rangle_{LP}^{L^P} &= \emptyset \\
\langle \alpha, n, H_{\text{pre}}, ak, \bar{f} \rangle_{LP}^{L^P} &= \begin{cases} 
\bar{s}, f; s, \bar{f} \mid s, ak', H', \bar{f}, f = \langle \alpha, n, H_{\text{pre}}, ak, \bar{f} \rangle_{LP}^{L^P} \\
\bar{s}, \bar{f} \end{cases} \\
\langle \circ \rangle_{LP}^{L^P \text{-listact-b}} &= \langle \circ \rangle_{LP}^{L^P \text{-listact-i}}
\end{align*}
\]

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 \( ak' \), \( H \) 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 \( \llbracket \cdot \rrbracket_{L_P}^{L_U} \)

\[ \llbracket \ell \rrbracket_{L_P}^{L_U} : I \times \pi \rightarrow \{ A \} \]

\[ \llbracket \hat{I}, \pi \rrbracket_{L_P}^{L_U} = \begin{cases} \hat{A} = A_{ske} \downarrow s, \pi & \text{for all } s, \pi \in \{ s, \pi \} \\ \text{where } \{ s, \pi \} = \llbracket \pi, 1, H_0, \emptyset, \text{main} \rrbracket_{L_P}^{L_U} \\ H_0 = 0 \mapsto 0 : k_{\text{root}} \\ A_{ske} = \llbracket \hat{T} \rrbracket_{L_P}^{L_U} \end{cases} \] (\( \llbracket \cdot \rrbracket_{L_P}^{L_U} \)-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.

\[ \llbracket \cdot \rrbracket_{L_P}^{L_U} : A \times s, \pi \rightarrow A \] (\( \llbracket \cdot \rrbracket_{L_P}^{L_U} \)-join)

\[ H; F_1; \cdots; F; \cdots; F_n \downarrow s, \pi; s, f = H; F_1; \cdots; F'; \cdots; F_n \downarrow s, \pi \]

where \( F = f(x) \mapsto s'; \text{return} ; \)

\( F' = f(x) \mapsto s; s'; \text{return} ; \)

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 \( \ell_i \)) that are sorted (guards with a test for \( \ell_i = 4 \) are before those with a test \( \ell_i = 5 \)).

4.2.5 Correctness of the Back-translation

Theorem 4 (\( \llbracket \cdot \rrbracket_{L_P}^{L_U} \) is correct).

\[ \forall \]

\[ \text{if } \Omega_0 \left( A \left[ \llbracket C \rrbracket_{L_P}^{L_U} \right] \right) \xrightarrow{\pi} \Omega \]

\[ \Omega \xhookrightarrow{\pi} \Omega' \]

\[ \hat{I} = \text{name}(A) \]

\[ \hat{\alpha} \equiv \alpha' \cdot \alpha ? \]

\[ \ell_i; \ell_{\text{glob}} \notin \beta \]

then \( \exists A \in \llbracket I, \pi \rrbracket_{L_P}^{L_U} \)

such that \( \Omega_0 (A \mid C) = \pi \xrightarrow{\pi} \Omega \)

and \( \pi \approx_\beta \pi \)

25
\[ \Omega \approx_{\beta} \Omega \]
\[ \Omega.H.\ell_i = |\pi| + 1 \]

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 `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 \( L_{\text{glob}} \) 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 back-translation.
5 The Source Language: \( L^* \)

This is an imperative, concurrent while language with monitors.

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

We highlight elements that have changed from \( L^U \).
5.1 Static Semantics of $L^\tau$

The static semantics follows these typing judgments.

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

5.1.1 Auxiliary Functions

We rely on these standard auxiliary functions: \text{names}(\cdot) extracts the defined names (e.g., function and interface names). \text{fv}(\cdot) returns free variables while \text{fn}(\cdot) returns free names (i.e., a call to a defined function). \text{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 .\text{funs}, .\text{intfs} and .\text{mon}.

We denote access to parts of $M$ with a dot notation, so $M.\Delta$ means $\Delta$ where $M = (\{\sigma\}, \sim, \sigma_0, \Delta, \sigma_c)$.

5.1.2 Typing Rules

\[ \vdash C \]
\[ C \equiv \Delta; F; I \quad C \vdash F : \text{UN} \quad \text{names}(F) \cap \text{names}(I) = \emptyset \quad \Delta \vdash \text{ok} \quad \vdash C : \text{UN} \]
\[ \vdash C : \text{UN} \]
\[ C \vdash F : \text{UN} \]
\[ F \equiv f(x : \text{UN}) \rightarrow s; \text{return;} \quad C, \Delta; x : \text{UN} \vdash s \]
\[ C \equiv \Delta; F; I \quad \forall f \in \text{fn}(s), f \in \text{dom}(C.\text{funs}) \lor f \in \text{dom}(C.\text{intfs}) \]
\[ C \vdash F : \text{UN} \]
\[ \Delta, \Gamma \vdash \diamond \]

\[ \vdash C : \text{UN} \]

\[ C \vdash F : \text{UN} \]

\[ F \equiv f(x : \text{UN}) \rightarrow s; \text{return;} \quad C, \Delta; x : \text{UN} \vdash s \]
\[ C \equiv \Delta; F; I \quad \forall f \in \text{fn}(s), f \in \text{dom}(C.\text{funs}) \lor f \in \text{dom}(C.\text{intfs}) \]
\[ C \vdash F : \text{UN} \]

\[ \Delta, \Gamma \vdash \diamond \]

\[ \vdash C : \text{UN} \]

\[ C \vdash F : \text{UN} \]

\[ F \equiv f(x : \text{UN}) \rightarrow s; \text{return;} \quad C, \Delta; x : \text{UN} \vdash s \]
\[ C \equiv \Delta; F; I \quad \forall f \in \text{fn}(s), f \in \text{dom}(C.\text{funs}) \lor f \in \text{dom}(C.\text{intfs}) \]
\[ C \vdash F : \text{UN} \]

\[ \Delta, \Gamma \vdash \diamond \]
<table>
<thead>
<tr>
<th>Rule</th>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset; \emptyset \vdash \top$</td>
<td>$T\mathcal{L}^e$-env-e</td>
<td>$\emptyset; \emptyset \vdash \top$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \top \quad x \notin \text{dom}(\Gamma)$</td>
<td>$T\mathcal{L}^e$-env-var</td>
<td>$\Delta, \Gamma; (x : \tau) \vdash \top$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \top$</td>
<td>$T\mathcal{L}^e$-env-loc</td>
<td>$\Delta, \Gamma \vdash \top$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-e</td>
<td>$\Delta, \Gamma, \Gamma; (x : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Delta, \Gamma \vdash \bot$</td>
</tr>
<tr>
<td>$\emptyset \vdash \bot$</td>
<td>$T\mathcal{L}^e$-env-e</td>
<td>$\emptyset \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-e</td>
<td>$\Delta, \Gamma, \Gamma; (x : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\emptyset \vdash \bot$</td>
<td>$T\mathcal{L}^e$-env-e</td>
<td>$\emptyset \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-e</td>
<td>$\Delta, \Gamma, \Gamma; (x : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\emptyset \vdash \bot$</td>
<td>$T\mathcal{L}^e$-env-e</td>
<td>$\emptyset \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-e</td>
<td>$\Delta, \Gamma, \Gamma; (x : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\emptyset \vdash \bot$</td>
<td>$T\mathcal{L}^e$-env-e</td>
<td>$\emptyset \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-e</td>
<td>$\Delta, \Gamma, \Gamma; (x : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\emptyset \vdash \bot$</td>
<td>$T\mathcal{L}^e$-env-e</td>
<td>$\emptyset \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta, \Gamma \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-e</td>
<td>$\Delta, \Gamma, \Gamma; (x : \tau) \vdash \bot$</td>
</tr>
<tr>
<td>$\Delta; (l : \tau) \vdash \bot$</td>
<td>$T\mathcal{L}^e$-safe-loc</td>
<td>$\Gamma; (l : \tau) \vdash \bot$</td>
</tr>
</tbody>
</table>

29
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 new, t terms where τ is different from UN.

\[ \Delta, \Gamma \vdash_{\text{UN}} e : \text{UN} \]

### (TUL^{\text{true}}-

\[ \Delta, \Gamma \vdash \text{true} : \text{UN} \]

### (TUL^{\text{false}}-

\[ \Delta, \Gamma \vdash \text{false} : \text{UN} \]

### (TUL^{\text{nat}}-

\[ \Delta, \Gamma \vdash n : \text{UN} \]

### (TUL^{\text{pair}}-

\[ \Delta, \Gamma \vdash e_1 : \text{UN}, e_2 : \text{UN} \]

### (TUL^{\text{de-reference}}-

\[ \Delta, \Gamma \vdash e : \text{e} : \text{UN} \]

### (TUL^{\text{op}}-

\[ \Delta, \Gamma \vdash \text{e} \oplus \text{e}' : \text{UN} \]

### (TUL^{\text{cmp}}-

\[ \Delta, \Gamma \vdash \text{e} \otimes \text{e}' : \text{UN} \]

### (TUL^{\text{skip}}-

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{skip} \]

### (TUL^{\text{sequence}}-

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{s}_a \]

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{s} \]

### (TUL^{\text{assign}}-

\[ \Delta, \Gamma \vdash_{\text{UN}} x : \text{UN} \]

\[ \Delta, \Gamma \vdash_{\text{UN}} e' : \text{UN} \]

### (TUL^{\text{if}}-

\[ \Delta, \Gamma \vdash_{\text{UN}} e : \text{Bool} \]

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{s}_1 \]

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{s}_a \]

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{if e then s}_r \text{ else s}_e \]

### (TUL^{\text{fork}}-

\[ C, \Delta, \Gamma \vdash_{\text{UN}} \text{s} \]

\[ C, \Delta, \Gamma \vdash_{\text{UN}} (\| \text{s}) \]
5.2 Dynamic Semantics of \( L^\tau \)

Function \( \text{mon-care}(\cdot) \) returns the part of a heap the monitor cares for (Rule \( L^\tau \)-Monitor-related heap). Rules \( L^\tau \)-Jump-Internal to \( L^\tau \)-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 \( L^\tau \)-Plug tells how to obtain a whole program from a component and an attacker. Rule \( L^\tau \)-Initial State tells the initial state of a whole program. Rule \( L^\tau \)-Initial-heap produces a heap that satisfies a \( \Delta \), initialised with base values. Rule \( L^\tau \)-Monitor Step tells when a monitor makes a single step given a heap.

\[
\text{mon-care}(\cdot)
\]

\[
\begin{align*}
\text{(L}\tau\text{-Monitor-related heap)} & \quad H' = \{\ell \mapsto v : \tau \mid \ell \mapsto v : \tau \in H\} \\
& \quad \vdash \text{mon-care}(H, \Delta) = H'
\end{align*}
\]

\[
\text{Helpers}
\]

\[
\begin{align*}
\text{(L}\tau\text{-Jump-Internal)} & \quad (f' \in I \land f \in I) \\
& \quad (f' \notin I \land f \notin I) \\
& \quad \vdash f, f' : \text{internal} \\
\text{(L}\tau\text{-Jump-IN)} & \quad f \in I \land f' \notin I \\
& \quad \vdash f, f' : \text{in} \\
\text{(L}\tau\text{-Jump-OUT)} & \quad f \notin I \land f' \in I \\
& \quad \vdash f, f' : \text{out} \\
\text{(L}\tau\text{-Plug)} & \quad A \equiv H; F; [] \\
& \quad \vdash C, F : \text{whole} \\
& \quad \Delta \vdash H_0 \\
& \quad \text{main}(x : \text{UN}) \Rightarrow s; \text{return}; \in F \\
& \quad A[C] = \Delta ; H \cup H_0 ; F ; F' ; I \\
\text{(L}\tau\text{-Whole)} & \quad C \equiv \Delta ; F ; I \\
& \quad \text{names}(F) \cap \text{names}(F') = \emptyset \\
& \quad \text{names}(I) \subseteq \text{names}(F) \cup \text{names}(F') \\
& \quad \vdash C, F : \text{whole} \\
\text{(L}\tau\text{-Initial State)} & \quad P \equiv \Delta ; H ; F ; I \\
& \quad C \equiv \Delta ; F ; I \\
& \quad \Omega_0 (P) = C, H \triangleright \text{call main 0} \\
\end{align*}
\]

\[
\text{(L}\tau\text{-Initial-heap)} \\
\Delta \vdash H \\
\emptyset \vdash v : \tau \\
\Delta, \ell : \tau \vdash H; \ell \mapsto v : \tau
\]

\[
\text{M;} H \rightsquigarrow M'
\]

32
5.2.1 Component Semantics

Expression $e$ reduces to $e'$.

Process $\pi$ reduces to $\pi'$ and evolves the rest accordingly.

Soup $\Pi$ reduce to $\Pi'$ and evolve the rest accordingly.

Program state $\Omega$ steps to $\Omega'$ emitting trace $\pi$.
6 \( L^\pi \): Extending \( L^P \) 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.

\[
\begin{align*}
\text{Whole Programs}\ P & ::= H_0; F; I \\
\text{Components}\ C & ::= H_0; F; I \\
\text{Statements}\ s & ::= \cdots \mid (\| s) \mid \text{destruct } x = e \text{ as } B \text{ in } s \text{ or } s \\
& \quad \mid \text{let } x = \text{newhide } e \text{ in } s \\
\text{Patterns}\ B & ::= \text{nat} \mid \text{pair} \\
\text{Monitors}\ M & ::= (\{\sigma\}, \Rightarrow, \sigma_0, H_0, \sigma_c) \\
\text{Single Process}\ \pi & ::= (s) \\
\text{Processes}\ \Pi & ::= \emptyset \mid \Pi \parallel \pi \\
\text{Prog. States}\ \Omega & ::= C, H \triangleright \Pi
\end{align*}
\]

6.2 Dynamic Semantics

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

\[
\begin{align*}
\textbf{mon-care}(\cdot) &\quad \text{(L}\pi\text{-Monitor-related heap)} \\
H' = \{n \mapsto v : \eta \mid n \in \text{dom}(H_0) \text{ and } n \mapsto v : \eta \in H\} \\
\textbf{mon-care}(H, H_0) &= H'
\end{align*}
\]

\[
\begin{align*}
\text{Helpers} &\quad \text{(L}\pi\text{-Plug)} \\
A \equiv F[\cdot] &\quad C \equiv H_0; F; I \\
\vdash C, F : \text{whole} &\quad \text{main}(x) \mapsto \text{return}; s \in F \\
C \vdash_{\text{att}} A &\quad \forall n \mapsto v : k \in H_0, k \in H_0 \\
A \mid C &\equiv H_0; F; F; I \\
(\text{L}\pi\text{-Initial State}) &\quad P \equiv H_0; F; I \\
\Omega_0(P) &= P, H_0 \triangleright \text{call main } 0
\end{align*}
\]

\[
\begin{align*}
M; H &\Rightarrow M'
\end{align*}
\]
6.2.1 Component Semantics

\[ C, H \triangleright \Pi \rightarrow C', H' \triangleright \Pi' \] Processes \( \Pi \) reduce to \( \Pi' \) and evolve the rest accordingly.
7 Extended Language Properties and Necessities

7.1 Monitor Agreement for \(L^\tau\) and \(L^\pi\)

Definition 16 (\(L^\tau\): \(M \prec C\)).

\[
(\{\sigma\}, \ni, \sigma_0, \Delta, \sigma_c) \prec (\Delta; F; I)
\]

A monitor and a component agree if they focus on the same set of locations \(\Delta\).

Definition 17 (\(L^\pi\): \(M \prec C\)).

\[
(\{\sigma\}, \ni, \sigma_0, H_0, \sigma_c) \prec (H_0; F; I)
\]

A monitor and a component agree if they focus on the same set of locations, protected with the same capabilities \(H_0\).

7.2 Properties of \(L^\tau\)

Definition 18 (\(L^\tau\) Semantics Attacker).

\[
\begin{align*}
C \vdash_{\text{attacker}} A & \coloneqq \forall \ell \in \text{dom}(C, \Delta), \ell \notin \text{fn}(A) \quad \text{no let } x = \text{new}, e \in s \text{ in } A \text{ such that } \tau \neq \text{UN} \\
C \vdash_{\text{att}} A & \coloneqq C = \Delta; F; I, \Delta \vdash_{\text{UN}} A \\
C \vdash_{\text{att}} \pi & \coloneqq \pi = (s)_f f \text{ and } f \in C.\text{its} \\
C \vdash_{\text{att}} \Pi \rightarrow \Pi' & \coloneqq \Pi = \Pi_1 \parallel \pi \parallel \Pi_2 \text{ and } \Pi' = \Pi_1 \parallel \pi' \parallel \Pi_2 \\
& \text{and } C \vdash_{\text{att}} \pi \text{ and } C \vdash_{\text{att}} \pi'
\end{align*}
\]

The two notions of attackers coincide.

Lemma 4 (Semantics and typed attackers coincide).

\[
C \vdash_{\text{attacker}} A \iff (C \vdash_{\text{att}} A)
\]

Theorem 5 (Typability Implies Robust Safety in \(L^\tau\)).

\[
\forall C, M \\
\text{if } \vdash C : \text{UN} \\
C \prec M \\
\text{then } M \vdash C : rs
\]
7.3 Properties of $L^π$

Definition 20 ($L^π$ Attacker).

$C \vdash \text{att } A \overset{\text{def}}{=} C = H_0; \mathcal{F}; \mathcal{I}, \forall k \in H_0. k \notin \text{fv}(A)$

$C \vdash \text{att } π \overset{\text{def}}{=} π = (s)_f, f \in C. \text{its}$

$C \vdash \text{att } \Pi \rightarrow \Pi' \overset{\text{def}}{=} \Pi = \Pi_1 || π || \Pi_2 \text{ and } \Pi' = \Pi_1 || π' || \Pi_2$

and $C \vdash \text{att } π \text{ and } C \vdash \text{att } π'$
8 Compiler from $L^\tau$ to $L^\pi$

8.1 Assumed Relation between $L^\tau$ and $L^\pi$ Elements

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

\[
M \approx M
\]

\[
(\text{Ok Mon})
\]
\[
M = (\{\sigma\}, \sim, \sigma_0, H_0, \sigma_c)
\]
\[
\forall \sigma \in \{\sigma\}, \forall \text{mon-care}(H; \Delta) \approx_\beta \text{mon-care}(H, H_0).
\]
\[
\text{if } H : \Delta \text{ then } \exists \sigma', (\sigma, \text{mon-care}(H, H_0), \sigma') \in \sim
\]

\[
(\text{Monitor relation})
\]
\[
\beta_0, \Delta \vdash M
\]
\[
\beta_0 = (\text{dom}(\Delta), \text{dom}(H_0), H_0, \eta)
\]

\[
\Delta \vdash H \quad \Delta, H \vdash v : \tau
\]

\[
(\text{Initial-heap})
\]
\[
\Delta \vdash H \quad \Delta, H \vdash \ell : \tau
\]
\[
\ell \approx_\beta (n, k)
\]

\[
(\text{Initial-values})
\]
\[
(\tau \equiv \text{Bool} \land v \equiv 0) \lor (\tau \equiv \text{Nat} \land v \equiv 0) \lor (\tau \equiv \text{Ref} \tau \land v \equiv n' \land n' \rightarrow v' : k' \in H \land \ell' \approx_\beta (n', k') \land \ell : \tau \in \Delta, \Delta, H \vdash v' : \tau)
\]

\[
\Delta, H \vdash v : \tau
\]

\[
(\text{Single process relation})
\]
\[
\tilde{T} \approx \tilde{T}
\]

\[
(\text{Process relation})
\]
\[
\tilde{T} \approx \tilde{T}
\]

\[
\Pi \approx_\beta \Pi
\]

\[
(\text{Process relation})
\]
\[
\Pi \approx_\beta \Pi
\]

\[
(\text{Process relation})
\]
\[
\Pi \parallel \Pi \approx_\beta \Pi \parallel \Pi
\]

8.2 Compiler Definition

Definition 21 (Compiler $L^\tau$ to $L^\pi$). \[| |]_{L^\tau}^{L^\pi} : C \rightarrow C \]
Given that $C = \Delta; F; I$ if $C : UN$ then $[C]_{L^T}$ is defined as follows:

\[
\begin{align*}
\frac{\text{(TL''-component)}}{C \equiv \Delta; F; I} & \quad L^T \\
\frac{C \vdash F : UN}{\text{names}(F) \cap \text{names}(I) = \emptyset} & \quad \Delta \vdash \text{ok} \\
\frac{\vdash C : UN}{\quad} & \quad \text{if } \Delta \vdash_{\beta_0} \text{H}_0 \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-function)}}{F \equiv f(x : UN) \mapsto s; \text{return}; C, \Delta; x : UN \vdash s} & \quad L^T \\
\frac{\forall f \in \text{fn}(s), f \in \text{dom}(C.funs)}{\forall f \in \text{dom}(C.intfs)} & \quad C \vdash F : UN \rightarrow UN \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-true)}}{\Delta, \Gamma \vdash \text{true} : \text{Bool}} & \quad L^T \\
\frac{\Delta, \Gamma \vdash \text{true} : \text{Bool}}{\quad} & \quad = 0 \quad \text{if } \text{true} \approx_{\beta} 0 \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-false)}}{\Delta, \Gamma \vdash \text{false} : \text{Bool}} & \quad L^T \\
\frac{\Delta, \Gamma \vdash \text{false} : \text{Bool}}{\quad} & \quad = 1 \quad \text{if } \text{false} \approx_{\beta} 1 \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-nat)}}{\Delta, \Gamma \vdash n : \text{Nat}} & \quad L^T \\
\frac{\Delta, \Gamma \vdash n : \text{Nat}}{\quad} & \quad = n \quad \text{if } n \approx_{\beta} n \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-var)}}{x : \tau \in \Gamma} & \quad L^T \\
\frac{\Delta, \Gamma \vdash x : \tau}{\quad} & \quad = x \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-loc)}}{\ell : \tau \in \Delta} & \quad L^T \\
\frac{\Delta, \Gamma \vdash \ell : \tau}{\quad} & \quad = \langle n, v \rangle \quad \text{if } \ell \approx_{\beta} \langle n, v \rangle \\
\end{align*}
\]

\[
\begin{align*}
\frac{\text{(TL''-pair)}}{\Delta, \Gamma \vdash e_1 : \tau} & \quad L^T \\
\frac{\Delta, \Gamma \vdash e_2 : \tau'}{\quad} & \quad \Delta, \Gamma \vdash \langle e_1, e_2 \rangle : \tau \times \tau' \\
\end{align*}
\]
We write wrong as a shortcut for a failing expression like 3 + true.

The remark about optimisation for \([\ell]_L^{\tau}\)-Endorse in Section 4 is also valid for the Rule ([\ell]_L^{\tau}-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 ([\ell]_L^{\tau}-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 ([\ell]_L^{\tau}-New) using non-atomic instructions are defined in Rule ([\ell]_L^{\tau}-New-nonat) be-
8.3 Properties of the $L^\tau$-$L^\pi$ Compiler

**Theorem 6** (Compiler $[[\cdot]]_{L^\pi}$ is CC). $\vdash [[\cdot]]_{L^\tau} : CC$

**Theorem 7** (Compiler $[[\cdot]]_{L^\pi}$ is RSC). $\vdash [[\cdot]]_{L^\tau} : RSC$

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

We define a more lenient relation on states $\equiv_{\beta}$ 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.

\[ \Omega \equiv_{\beta} \Omega \]

\[
\begin{align*}
\vdash \text{secure}(H) = H' \\
\# \ell \in \text{secure}(H) & \quad \ell \approx_{\beta} (n,_) & n \in \text{dom}(H) \\
H, H \vdash \text{low-loc}(n) \\
\ell \in \text{secure}(H) & \quad \ell \approx_{\beta} (n,k) & n \mapsto _k : k \in H \\
H, H \vdash \text{high-loc}(n) = \ell, k \\
H, H \vdash \text{high-cap}(k)
\end{align*}
\]
There is no $\text{secure}(\cdot)$ 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 $\text{secure}(\cdot)$ as all locations protected by a key $k$ but the point of $\text{secure}(\cdot)$ is to setup the invariant to ensure the proof hold, so this alternative would be misleading.

Rule $L^\pi$-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 $\beta$ 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 $n$ that is high (i.e., it has a related source counterpart $\ell$ whose type is secure and that is protected with a capability $k$ that we call a high capability), then we have: (1) the capability $k$ used to lock it is not in any attacker code; (2) for any target level location $n'$: (2a) either it is locked with a high capability $k$ (i.e., a capability used to hide a high location) thus $n'$ is also high, in which case it is related to a source location $\ell$ and the values $v, v$ they point to are related; or (2b) it is not locked with a high capability, so we can derive that $n'$ is a low location and its content $v$ is not any high capability $k'$.

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

\[ \forall n \in H \implies \text{low-loc}(n) \lor \exists \ell \in \text{dom}(H). \text{high-loc}(n) = \ell, \eta \]

Proof. Trivial, as Rule $L^\pi$-Low Location and Rule $L^\pi$-High Location are duals. \( \square \)
9 \textit{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 \textit{RSC} is only useful for this kind of a target, we show here how to attain \textit{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 \( \llbracket J \rrbracket_L^\tau \mapsto_L L^I \) from our source language \( L^\tau \) to \( L^I \) and why it attains \textit{RSC}. Full formal details are presented in subsequent sections.

9.1 \( L^I \), a Target Language with Memory Isolation

Language \( L^I \) replaces \( L^\tau \)'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 \( C ::= H_0 ; \overline{E} ; T ; \overline{E} \). Only functions that are listed in \( \overline{E} \) can create \((\text{let } x = \text{newiso } e \text{ in } s)\), 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 \( L^\tau \), but the expression semantics change to \( C ; H ; f \triangleright e \iff v \), 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 \( C \vdash f : \text{progs} \)). Monitors of \( L^I \) are the same as those of \( L^\tau \).

9.2 Compiler from \( L^\tau \) to \( L^I \)

The high-level structure of the compiler \( \llbracket \_ \rrbracket_L^\tau \mapsto_L L^I \) is similar to that of \( \llbracket \_ \rrbracket_L^\tau \mapsto_L L^\tau \). \( \llbracket \_ \rrbracket_L^\tau \) 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 \( \Delta \) (captured by the judgement \( \Delta \vdash H_0 \), whose details we elide here). Importantly, \( \llbracket \_ \rrbracket_L^I \) 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 $\text{UN}$) are placed outside the enclave while those that trusted (not subtypes of $\text{UN}$) are placed inside.

As mentioned, $\llbracket \cdot \rrbracket^I_T$, 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

Whole Programs $P ::= H_0; F; I; E$

Components $C ::= H_0; F; I; E$

Contexts $A ::= F[\cdot ]$

Interfaces $I ::= f$

Enclave functions $E ::= f$

Functions $F ::= f(x) \mapsto s; \text{return}$

Operations $\oplus ::= + | −$

Comparison $\otimes ::= == | < | >$

Values $v ::= n \in \mathbb{Z} | \langle v, v \rangle | k$

Expressions $e ::= x | v | e \oplus e | e \otimes e | \langle e, e \rangle | e.1 | e.2 | !e$

Statements $s ::= \text{skip} | s ; s | \text{let } x = e \text{ in } s | \text{if } e \text{ then } s \text{ else } s | \text{call } f e$

| $| | (\langle s \rangle ) | \text{destruct } x = e \text{ as } B \text{ in } s \text{ or } s$

| $x ::= e | \text{let } x = \text{new } e \text{ in } s | \text{let } x = \text{newiso } e \text{ in } s$

Patterns $B ::= \text{nat} | \text{pair}$

Eval. Ctxs. $E ::= [\cdot ] | e \oplus E | E \oplus n | e \otimes E | E \otimes n | !E$

| $\langle e, E \rangle | \langle E, v \rangle | E.1 | E.2$

Heaps $H ::= \emptyset | H; n \mapsto v$

Monitors $M ::= (\{\sigma\}; \rightsquigarrow; \sigma_0, H_0, \sigma_c)$

Mon. States $\sigma \in S$

Mon. Reds. $\rightsquigarrow ::= \emptyset | \rightsquigarrow; (s, H, s)$

Substitutions $\rho ::= \emptyset | \rho[v / x]$

Single Process $\pi ::= (s)_T$

Processes $\Pi ::= \emptyset | \Pi \parallel \pi$

48
10.2 Operational Semantics of \( L^I \)

10.2.1 Component Semantics

\[
C; H; f \triangleright e \rightsquigarrow e'
\]
Expression \( e \) reduces to \( e' \).

\[
C; H \triangleright \Pi \rightsquigarrow C'; H' \triangleright \Pi'
\]
Processes \( \Pi \) reduce to \( \Pi' \) and evolve the rest accordingly, emitting label \( \lambda \).

\[
\Omega \xrightarrow{\pi} \Omega'
\]
Program state \( \Omega \) steps to \( \Omega' \) emitting trace \( \pi \).

\[
\Omega \xrightarrow{\pi} \Omega'
\]
We elide the suffix with the stack of functions when obvious.
(E₁⁻assign-hoc)

\[ C; H; \overline{f} \triangleright e \iff v \quad \overline{f} = \overline{f'} \cdot f \quad C \vdash f : \text{prog} \]

\[ H = H'_1; n \mapsto \_; H'_2 \quad H'_1 = H'; n \mapsto v; H'_2 \quad n < 0 \]

\[ C, H \triangleright (n := e) \rightarrow C, H' \triangleright \text{(skip)} \]

\[ (E₁⁻call-internal) \]

\[ \overline{C}.\text{intfs} \vdash f, f' : \text{internal} \quad \overline{f'} = \overline{f''}; f' \]

\[ f(x) \mapsto s; \text{return}; \in \overline{C}.\text{funs} \quad C, H; \overline{f} \triangleright e \iff v \]

\[ C, H \triangleright (\text{call}\ f\ e) \rightarrow C, H \triangleright (s; \text{return};[v / x]) \]

\[ (E₁⁻callback) \]

\[ \overline{C}.\text{intfs} \vdash f', f : \text{out} \quad C, H; \overline{f} \triangleright e \iff v \]

\[ C, H \triangleright (\text{call}\ f\ e) \rightarrow C, H \triangleright (s; \text{return};[v / x]) \]

\[ (E₁⁻call) \]

\[ \overline{C}.\text{intfs} \vdash f', f : \text{in} \quad C, H; \overline{f} \triangleright e \iff v \]

\[ C, H \triangleright (\text{call}\ f\ e) \rightarrow C, H \triangleright (s; \text{return};[v / x]) \]

\[ (E₁⁻res-internal) \]

\[ \overline{C}.\text{intfs} \vdash f, f' : \text{internal} \quad \overline{f} = \overline{f''}; f' \]

\[ C, H \triangleright (\text{return};) \rightarrow C, H \triangleright \text{(skip)} \]

\[ (E₁⁻ret-back) \]

\[ \overline{C}.\text{intfs} \vdash f, f' : \text{in} \quad \overline{f} = \overline{f''}; f' \]

\[ C, H \triangleright (\text{return};) \rightarrow C, H \triangleright \text{(skip)} \]

\[ (E₁⁻return) \]

\[ \overline{C}.\text{intfs} \vdash f, f' : \text{out} \quad \overline{f} = \overline{f''}; f' \]

\[ C, H \triangleright (\text{return};) \rightarrow C, H \triangleright \text{(skip)} \]

\[ (E₁⁻destruct-nat) \]

\[ C; H; \overline{f} \triangleright e \iff n \]

\[ C, H \triangleright \text{destruct } x = e \text{ as } n \text{ in } s \text{ or } s' \rightarrow C, H \triangleright s[n / x] \]

\[ (E₁⁻destruct-pair) \]

\[ C; H; \overline{f} \triangleright e \iff (v, v') \]

\[ C, H \triangleright \text{destruct } x = e \text{ as pair in } s \text{ or } s' \rightarrow C, H \triangleright s [(v, v') / x] \]

\[ (E₁⁻destruct-nat) \]

\[ \text{otherwise} \]

\[ C, H \triangleright \text{destruct } x = e \text{ as } B \text{ in } s \text{ or } s' \rightarrow C, H \triangleright s' \]

\[ C, H \triangleright \Pi \iff C', H' \triangleright \Pi' \]

51
\[ \Pi = \Pi_1 \parallel (s) \parallel \Pi_2 \]
\[ \Pi' = \Pi_1 \parallel (s') \parallel \Pi_2 \]
\[ C, H \triangleright (s) \iff C', H' \triangleright (s') \]
\[ C, H \triangleright \Pi \iff C', H' \triangleright \Pi' \]

\[ \Omega \xrightarrow{\text{[EOLI-par]}} \Omega' \]

\[ \begin{align*}
\[ \Omega \xrightarrow{\text{[EOLI-single]}} \Omega'' \] & \quad \Omega'' \xrightarrow{\text{[EOLI-trans]}} \Omega' \\
\[ \Omega \xrightarrow{\text{[EOLI-empt]}} \Omega' \] & \quad \Omega \xrightarrow{\text{[EOLI-empt]}} \Omega'' \\
\end{align*} \]

10.3 Monitor Semantics

\[ \text{mon-care}(\cdot) \]

\[ H' = \{ n \mapsto v : \eta \mid n \in \text{dom}(H_0) \text{ and } n \mapsto v : \eta \in H \} \]

\[ \text{mon-care}(H, H_0) = H' \]

\[ \begin{align*}
M; H \rightsquigarrow M' \\
\end{align*} \]

\[ \begin{align*}
\begin{align*}
\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c & \quad M' = \{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_j \\
\end{align*} \\
\begin{align*}
(s, \text{mon-care}(H, H_0), s_j) & \in \rightsquigarrow \\
\end{align*} \\
\begin{align*}
\begin{align*}
M; H \rightsquigarrow M' \\
\end{align*} \\
\begin{align*}
\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c : (H_0; \overline{F}; \overline{T}; \overline{E}) \\
\end{align*} \]

10.4 Monitor Agreement for $L^I$

Definition 22 ($L^I$: $M \bowtie C$).

\[ (\{\sigma\}, \rightsquigarrow, \sigma_0, H_0, \sigma_c) \bowtie (H_0; \overline{F}; \overline{T}; \overline{E}) \]

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

52
10.5 Properties of $L^I$

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

\[
C \vdash_{\text{att}} A \overset{\text{def}}{=} C = H_0; F; I; E, A = F', \text{names}(F) \cap \text{names}(F') = \emptyset
\]
\[
C \vdash_{\text{att}} \pi \overset{\text{def}}{=} (s)_{\text{itfs}}^f \text{ and } f \in C.\text{itfs}
\]
\[
C \vdash_{\text{att}} II \rightarrow II' \overset{\text{def}}{=} I_I \parallel \pi \parallel II_2 \text{ and } II' = I_I \parallel \pi' \parallel II_2
\]
\[
\text{and } C \vdash_{\text{att}} \pi \text{ and } C \vdash_{\text{att}} \pi'
\]

11 Second Compiler from $L^\tau$ to $L^I$

For this compiler we need a different partial bijection, which we indicate with \(\varphi\) and its type is \(\ell \times n\). It has the same properties of \(\beta\) 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 \(M \approx M\) is defined as in Rule Monitor relation.

Some auxiliary functions are changed:

\[
\Delta \vdash_{\varphi} H_0 \quad \Delta, H \vdash v : \tau
\]

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

\[
(\text{initial-value})
\tau \equiv \text{Ref} \land v \equiv n' \land n' \mapsto v' \in H \land \ell' \approx_{\varphi} n' \land \ell : \tau \in \Delta, \Delta, H \vdash v' : \tau
\]
\[
(\tau \equiv \tau_1 \times \tau_2 \land v \equiv (v_1, v_2) \land \Delta, H \vdash v_1 : \tau_1 \land \Delta, H \vdash v_2 : \tau_2)
\]
\[
\Delta, H \vdash_{\varphi} v : \tau
\]

**Definition 24 (Compiler $L^\tau$ to $L^I$).** $\llbracket L^I \rrbracket : C \rightarrow C$
Given that $C = \Delta; \Gamma; \Gamma$ if $\vdash C : \text{UN}$ then $\llbracket C \rrbracket^L_T$ is defined as follows:

\[
\begin{align*}
(\text{TL}^\tau\text{-component}) & \quad \llbracket C \rrbracket^L_T = H_0; \llbracket F \rrbracket^L_T; \llbracket \Pi \rrbracket^T_{L/I}; \text{dom}(F) & \quad \text{if } \Delta \vdash_{\phi_0} H_0 \\
(\text{TL}^\tau\text{-function}) & \quad \llbracket F \rrbracket^L_T = f(x : \text{UN}) \mapsto s; \text{return; } & \quad \text{if } \Delta \vdash_{\phi_0} C : \text{UN} \\
& \quad \forall f \in \text{fn}(s), f \in \text{dom}(C.\text{funs}) \forall f \in \text{dom}(C.\text{intfs}) & \text{if } \Delta \vdash_{\phi_0} C : \text{UN} \\
& \quad \Gamma \vdash F : \text{UN} \rightarrow \text{UN} & \text{if } \Delta \vdash_{\phi_0} C : \text{UN}
\end{align*}
\]

Expressions

\[
\begin{align*}
(\text{TL}^\tau\text{-true}) & \quad \Delta, \Gamma \vdash \text{true} : \text{Bool} & \quad \llbracket \text{true} \rrbracket^L_T = 0 & \quad \text{if } \text{true} \approx_{\phi_0} 0 \\
(\text{TL}^\tau\text{-false}) & \quad \Delta, \Gamma \vdash \text{false} : \text{Bool} & \quad \llbracket \text{false} \rrbracket^L_T = 1 & \quad \text{if } \text{false} \approx_{\phi_0} 1 \\
(\text{TL}^\tau\text{-nat}) & \quad \Delta, \Gamma \vdash n : \text{Nat} & \quad \llbracket n \rrbracket^L_T = n & \quad \text{if } n \approx_{\phi_0} n \\
(\text{TL}^\tau\text{-var}) & \quad \Delta, \Gamma \vdash x : \tau & \quad \llbracket x \rrbracket^L_T = x & \quad \text{if } \Delta \vdash_{\phi_0} x : \tau \\
(\text{TL}^\tau\text{-loc}) & \quad \Delta, \Gamma \vdash \ell : \tau & \quad \llbracket \ell \rrbracket^L_T = n & \quad \text{if } \Delta \vdash_{\phi_0} \ell \approx_{\phi_0} n \\
(\text{TL}^\tau\text{-pair}) & \quad \Delta, \Gamma \vdash e_1 : \tau, \Delta, \Gamma \vdash e_2 : \tau' & \quad \llbracket \langle \Delta, \Gamma \vdash e_1 \vdash \tau, \Delta, \Gamma \vdash e_2 \vdash \tau' \rangle \rrbracket^L_T & \quad \text{if } \Delta \vdash_{\phi_0} (e_1, e_2) : \tau \times \tau'
\end{align*}
\]

54
\[
\begin{align*}
&\frac{(\text{TL}^\tau\text{-proj-1})}{\Delta, \Gamma \vdash e : \tau \times \tau'}{\Delta, \Gamma \vdash e.1 : \tau} \quad (\text{L}^\tau\text{-P1}) \\
&\frac{(\text{TL}^\tau\text{-proj-2})}{\Delta, \Gamma \vdash e : \tau \times \tau'}{\Delta, \Gamma \vdash e.2 : \tau'} \quad (\text{L}^\tau\text{-P2}) \\
&\frac{(\text{TL}^\tau\text{-dereference})}{\Delta, \Gamma \vdash e : \text{Ref} \tau}{\Delta, \Gamma \vdash \text{Ref} e : \tau} \quad (\text{L}^\tau\text{-Deref}) \\
&\frac{(\text{TL}^\tau\text{-op})}{\Delta, \Gamma \vdash e : \text{Nat}}{\Delta, \Gamma \vdash e' : \text{Nat}} \quad (\text{L}^\tau\text{-op}) \\
&\frac{(\text{TL}^\tau\text{-cmp})}{\Delta, \Gamma \vdash e : \text{Nat}}{\Delta, \Gamma \vdash e' : \text{Nat}} \quad (\text{L}^\tau\text{-cmp}) \\
&\frac{(\text{TL}^\tau\text{-coerce})}{\Delta, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash e : \un} \quad (\text{L}^\tau\text{-Coerce}) \\
&\frac{(\text{TL}^\tau\text{-skip})}{\text{C}, \Delta, \Gamma \vdash \text{skip}}{\text{L}^\tau\text{-Skip}} \\
&\frac{(\text{TL}^\tau\text{-new})}{\text{C}, \Delta, \Gamma \vdash e : \tau}{\text{C}, \Delta, \Gamma ; x : \text{Ref} \tau \vdash s} \quad (\text{L}^\tau\text{-New})
\end{align*}
\]
\[ \begin{array}{c}
\text{(TL-endomorphism)} \\
\Delta, \Gamma \vdash e : \text{UN} \\
\text{C, } \Delta, \Gamma : \phi \vdash s
\end{array} \quad \begin{array}{c}
\text{C, } \Delta, \Gamma \vdash \text{endorse } x = e \text{ as } \phi \text{ in } s
\end{array} \]

\begin{equation}
\begin{cases}
\text{destruct } x = \begin{cases}
\Delta, \Gamma \vdash e : \text{UN}^L_L, \text{as nat in if } x \text{ then} \\
\text{C, } \Delta, \Gamma : \phi \vdash s^L_L \\
\text{else if } x - 1 \text{ then} \\
\text{C, } \Delta, \Gamma : \phi \vdash s^L_L \\
\text{else wrong or wrong}
\end{cases}
\end{cases}
\end{equation}

We use \textit{wrong} as before for \textit{wrong}.

11.1 Properties of the \(L^L-L_J\) Compiler

Theorem 8 (Compiler \(\llbracket \cdot \rrbracket_L^L\) is CC), \(\vdash \llbracket \cdot \rrbracket_L^L : CC\)

Theorem 9 (Compiler \(\llbracket \cdot \rrbracket_L^L\) is RSC), \(\vdash \llbracket \cdot \rrbracket_L^L : RSC\)

11.2 Cross-language Relation \(\equiv_{\phi}\)

As before, we define a more lenient relation on states \(\equiv_{\phi}\)

\[ \Omega \equiv_{\phi} \Omega \]

\begin{equation}
\begin{array}{c}
(\llbracket L\text{-Low Location} \rrbracket) \\
\bar{\ell} \in \text{secure}(H) \quad \ell \approx_{\phi} n \quad n \geq 0
\end{array}
\end{equation}

\[ H, H \vdash \text{low-loc}(n) \]

\begin{equation}
\begin{array}{c}
(\llbracket L\text{-High Location} \rrbracket) \\
\ell \in \text{secure}(H) \quad \ell \approx_{\phi} n \quad n < 0
\end{array}
\end{equation}

\[ H, H \vdash \text{high-loc}(n) = \ell \]

57
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).

\[\forall n, \ell. \text{if } H \approx_{\phi} H \\Rightarrow \begin{cases} H, H \vdash \text{low-loc}(n) \\ \exists \ell \in \text{dom}(H). \quad H, H \vdash \text{high-loc}(n) = \ell \end{cases}\]

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

12.1 Proof of Theorem 1 (\textit{PF-RSC and RSC are equivalent})

\textit{Proof.} \Rightarrow \text{HP}

\[
\text{if } \forall A, \pi. [C]_T \vdash A : \text{attacker} \\
\quad \vdash A [C]_T : \text{whole } \Omega_0 ([C]_T) \xRightarrow{\pi} \Omega \\
\text{then } \exists A, \pi C \vdash A : \text{attacker} \quad \vdash A [C] : \text{whole } \Omega_0 (A [C]) \xRightarrow{\pi} \Omega \\
\quad \text{relevant}(\pi) \approx_{\beta} \text{relevant}(\pi)
\]

\textit{TH}

\[
\text{if } M \approx_{\beta} M \\
\quad \forall A, \pi. \vdash A [C] : \text{whole } \quad \text{if } \Omega_0 (C) \xRightarrow{\pi} \Omega \\
\text{then } M \vdash \pi
\]

\[
\text{then } \forall A, \pi. \vdash A [C]_T : \text{whole } \quad \text{if } \Omega_0 ([C, M]_T) \xRightarrow{\pi} \Omega \\
\text{then } M \vdash \pi
\]

We proceed by contradiction and assume that \( M \not\approx_{\beta} \pi \) while \( C \vdash \pi \).

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

But if the heaps are related and the source steps (by unfolding \( M \vdash \pi \)), then by point 3.b in Definition 8 (\( MRM \)) we have that the target monitor also steps, so \( M \vdash \pi \).

We have reached a contradiction, so this case holds.

\( \Leftarrow \) Switch HP and TH from the point above.

Analogously, we proceed by contradiction:

\[
\quad \forall A, \pi. \vdash A [C] : \text{whole } \quad \text{and } \Omega_0 (A [C]) \xRightarrow{\pi} \Omega \text{ and } \text{relevant}(\pi) \not\approx_{\beta} \text{relevant}(\pi)
\]

By the same reasoning as above, with the HP we have we obtain \( M \vdash \pi \) and \( M \vdash \pi \).

Again by 3.b in Definition 8 (\( MRM \)) we know that the heaps of all actions in the traces are related.

Therefore, \( \text{relevant}(\pi) \approx_{\beta} \text{relevant}(\pi) \), which gives us a contradiction. \( \square \)
12.2 Proof of Theorem 2 (Compiler $\llbracket \cdot \rrbracket^{LU}_{LP}$ is CC)

Proof. The proof proceeds for $\beta_0 = (\ell, 0, k_{\text{root}})$ and then, given that the languages are deterministic, by Lemma 8 (Generalised compiler correctness for $\llbracket \cdot \rrbracket^{LU}_{LP}$) as initial states are related by definition. □

Lemma 7 (Expressions compiled with $\llbracket \cdot \rrbracket^{LU}_{LP}$ are related).

$$\forall$$

if $H \equiv_{\beta} H$

$H \triangleright e_\rho \iff v$

then $H \triangleright [e]^{LU}_{LP} [\beta]^{LU}_{LP} \iff [v]^{LU}_{LP}$

Proof. This proof proceeds by structural induction on $e$.

Base case: Values

true By Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-True), $\llbracket \text{true} \rrbracket^{LU}_{LP} = 0$.

As true $\equiv_{\beta} 0$, this case holds.

false Analogous to the first case by Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-False).

$n \in \mathbb{N}$ Analogous to the first case by Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-nat).

$x$ Analogous to the first case, by Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-Var) and by the relatedness of the substitutions.

$\ell$ Analogous to the first case by Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-Loc).

$\langle v, v \rangle$ By induction on $v$ by Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-Pair) and then it is analogous to the first case.

Inductive case: Expressions

$e \oplus e'$ By Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-op) we have that

$[e \oplus e']^{LU}_{LP} = [e]^{LU}_{LP} \oplus [e']^{LU}_{LP}$

By HP we have that $H \triangleright e_\rho \iff n$ and $H \triangleright e'_{\rho} \iff n'$.

By Rule EL$^{LU}_{LP}$-op we have that $H \triangleright n \oplus n' \iff n''$.

By IH we have that $H \triangleright [e]^{LU}_{LP} [\beta]^{LU}_{LP} \iff [n]^{LU}_{LP}$ and $H \triangleright [e']^{LU}_{LP} [\beta]^{LU}_{LP} \iff [n']^{LU}_{LP}$.

By Rule EL$^{LU}_{LP}$-op we have that $H \triangleright [n]^{LU}_{LP} \oplus [n']^{LU}_{LP} \iff [n'']^{LU}_{LP}$.

So this case holds.

$e \otimes e'$ Analogous to the case above by IH, Rule ($\llbracket \cdot \rrbracket^{LU}_{LP}$-cmp), Rule EL$^{LU}_{LP}$-comp and Rules EL$^{LU}_{LP}$-op and EL$^{LU}_{LP}$-if-true.
Analogous to the case above by IH twice, Rule $\text{deref}$, Rule $\text{E}-\text{deref}$ and Rules $\text{E}-\text{p1}$, $\text{E}-\text{p2}$ and $\text{E}-\text{letin}$ and a case analysis by Rules $\text{E}-\text{deref-top}$ and $\text{E}-\text{deref-k}$.

$\langle e, e \rangle$ Analogous to the case above by IH and Rule $\text{E}-\text{pair}$.

$(e, 1)$ By Rule $\text{E}$-\text{P1} $\exists e \approx e.$

By IH we have that $H \triangleright \langle e_1, e_2 \rangle \leadsto \langle e_1, e_2 \rangle.$

By Rule $\text{E}-\text{Pair}$ we have that $\langle e_1, e_2 \rangle \approx \langle e_1, e_2 \rangle.$

Now $H \triangleright \langle e_1, e_2 \rangle \leadsto \langle e_1, e_2 \rangle.$ So this case holds.

$(e, 2)$ Analogous to the case above by Rule $\text{E}$-\text{P2}, Rule $\text{E}$-\text{p2} and Rule $\text{E}$-\text{p2}.

\[ \Box \]

Lemma 8 (Generalised compiler correctness for $\text{E}$-\text{pair}).

Proof.

\[ \forall \ldots \exists \beta' \]

if $\vdash C : \text{whole}$

\[ C = \Delta : F ; I \]

\[ \| C \|_{L^r} = k_{\text{root}} : F ; I = C \]

\[ C, H \triangleright \| s \| \approx \beta C, H \triangleright \| s \|_{L^r} \]

\[ C, H \triangleright s \rho \leadsto C', H' \triangleright \| s \|_{L^r} \approx \beta' \]

then $C, H \triangleright \| s \|_{L^r} \approx \beta' C, H \triangleright \| s \|_{L^r} \approx \beta'_{L^u} \approx \beta'_{L^u}.$

\[ C = k_{\text{root}} : F ; I \]

\[ C, H \triangleright s' \approx \beta' C, H \triangleright \| s' \|_{L^r} \approx \beta'_{L^u} \approx \beta' \]

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

Base case

\[ \text{skip} \] By Rule $\text{E}$-\text{Skip} this case follows trivially.

Inductive
let x = new e in s

By Rule (\[ \{ \]_L^e-P-New)  \[ \{ \]_L^e =

let x_{loc} = new \[ e \]_L^e in
let x_{cap} = hide x_{loc} in
let x = \langle x_{loc}, x_{cap} \rangle in \[ s \]_L^e

By HP  H \triangleright e? \iff v

So by Lemma 7 we have HPE:  H \triangleright \[ e \]_L^e \ triangleright \[ v \]_L^e and HPV
\[ v \approx \beta \triangleright \[ v \]_L^e.

By Rule E\[ U\]_alloc:  C; H \triangleright x = new e in s \rightarrow C; H; \ell \rightarrow v \triangleright s[\ell / x].

So by HPE:

\[ C; H \triangleright x_{loc} = new \[ e \]_L^e in \\
let x_{cap} = hide x_{loc} in \\
let x = \langle x_{loc}, x_{cap} \rangle in \[ s \]_L^e \rho \]

Rule E\[ L\]_P-new

\[ \rightarrow C; H; n \rightarrow \[ v \]_L^e : \bot \triangleright let x_{cap} = hide x_{loc} in \\
let x = \langle x_{loc}, x_{cap} \rangle in \[ s \]_L^e \rho \]

\[ C; H; n \rightarrow \[ v \]_L^e : \bot \triangleright let x_{cap} = hide n in \\
let x = \langle n, x_{cap} \rangle in \[ s \]_L^e \rho \]

Rule E\[ L\]_P-hide

\[ \rightarrow C; H; n \rightarrow \[ v \]_L^e : k; k \triangleright let x = \langle n, x_{cap} \rangle in \[ s \]_L^e \rho \]

\[ C; H; n \rightarrow \[ v \]_L^e : k; k \triangleright \langle n, k \rangle in \[ s \]_L^e \rho \]

Rule E\[ L\]_P-letin

\[ \rightarrow C; H; n \rightarrow \[ v \]_L^e : k; k \triangleright \[ s \]_L^e \rho \]

Let \[ \beta' = \beta \cup (\ell, n, k). \]

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

By a simple weakening lemma for \[ \beta \] for substitutions and values applied to HP and HPV we can get HPBV  \[ v \approx_{\beta'} \[ v \]_L^e. \]

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

We have that \[ \rho = \rho[\ell / x] \] and \[ \rho' = \rho[n, k / x]. \]

So by HPL we get that \[ \rho' \approx_{\beta'} \rho'. \]
s; s′ Analogous to the case above by IH, Rule (\([U]_{LP}\text{-Seq}\)) and a case analysis on what s reduces to, either with Rule \([U]_{LP}\text{-sequence}\) and Rule \([P]_{LP}\text{-sequence}\) or with Rule \([U]_{LP}\text{-step}\) and Rule \([P]_{LP}\text{-step}\).

let x = e in s Analogous to the case above by IH, Rule (\([U]_{LP}\text{-letin}\)), Rule \([U]_{LP}\text{-letin}\), and Rule \([P]_{LP}\text{-letin}\).

x := e′ Analogous to the case above by Rule (\([U]_{LP}\text{-Assign}\)), Rule \([U]_{LP}\text{-update}\) and Rule \([P]_{LP}\text{-letin}\) (twice), Rules \([P]_{LP}\text{-p1}\) and \([P]_{LP}\text{-p2}\) and then a case analysis by Rules \([P]_{LP}\text{-assign-top}\) and \([P]_{LP}\text{-assign-k}\).

if e then s else s′ Analogous to the case above by IH, Rule (\([U]_{LP}\text{-If}\)) and then either Rule \([U]_{LP}\text{-if-true}\) and Rule \([P]_{LP}\text{-if-true}\) or Rule \([U]_{LP}\text{-if-false}\) and Rule \([P]_{LP}\text{-if-false}\).

call f e By Rule (\([U]_{LP}\text{-call}\)) \([call f e]_{LP} = call f [e]_{LP}\).

By HP \(\text{IH} \rightarrow \rho \mapsto v\).

So by Lemma 7 we have HPE: \(\text{IH} \rightarrow [e]_{LP} \rightarrow [v]_{LP}\) and HPR \(v \approx [v]_{LP}\).

So as \(C\) is whole, we apply Rule \([P]_{LP}\text{-call-internal}\)

\[
C, \text{IH} \vdash (\text{IH} \rightarrow \rho \mapsto v)_{\tau,f} \\
C, \text{IH} \vdash (s; \text{return};\rho[v/x])_{\tau,f}
\]

By Rule \([P]_{LP}\text{-call-internal}\)

\[
C, \text{IH} \vdash (\text{IH} \rightarrow \rho \mapsto [v]_{LP})_{\tau,f} \\
C, \text{IH} \vdash (s; \text{return};\rho[v/x])_{\tau,f}
\]

By the first induction on \(C\) we get

IH1 \(C, \text{IH} \vdash (s; \text{return};\rho')_{\tau,f} \approx_{\beta} C, \text{IH} \vdash (s; \text{return};\rho')_{\tau,f}\).

We instantiate \(\rho'\) with \(\rho[v/x]\) and \(\rho'\) with \([\rho']_{LP} [v]_{LP} / x\).

So by HP and HPR we have that \(\rho[v/x] \approx_{\beta} [\rho']_{LP} [v]_{LP} / x\).

We can use IH1 to conclude

\[
C, \text{IH} \vdash (s; \text{return};\rho[v/x])_{\tau,f} \approx_{\beta} C, \text{IH} \vdash (s; \text{return};\rho'[v/x])_{\tau,f}
\]

As \(\beta' = \beta\), this case holds.

\(\square\)

\(\bigstar\)
12.3 Proof of Theorem 3 (Compiler $\llbracket \cdot \rrbracket_{LP}^{LU}$ is $RSC$)

Proof. HPM: $M \approx_\beta M$

HP1: $M \vdash C : rs$

TH1: $M \vdash \llbracket CLP \rrbracket_{LU} : rs$

We can state it in contrapositive form as:

HP2: $M \not\vdash \llbracket CLP \rrbracket_{LU} : rs$

TH2: $M \not\vdash C : rs$

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

HP21 $\exists A, \pi. M \vdash A : \text{attacker}$ and either $\not\vdash A [JCLP] : \text{whole}$ or

HPR1 ($\Omega_0 (A [CLP]) \Rightarrow_\pi \Omega$) and HPRMT1 $M \not\approx \pi$

TH21 $\exists A, \pi. M \vdash A : \text{attacker}$ and either $\not\vdash A [C] : \text{whole}$ or

TH2 ($\Omega_0 (A [C]) \Rightarrow_\pi \Omega$) and TH4 $M \not\approx \pi$

We consider the case of a whole $A$, the other is trivial.

We can apply Theorem 4 ($\llbracket \cdot \rrbracket_{LP}$ is correct) with HPR1 and instantiate $A$ with a $A$ from $\llbracket A \rrbracket$ and we get the following unfolded HPs

HPRS $\Omega_0 (A [C]) \Rightarrow_\pi \Omega$

HPRel $\pi \approx_\beta \pi$

So TH3 holds by HPRS.

We need to show TH4

Assume by contradiction HPBOT: the monitor in the source does not fail:

$M \vdash \pi$

By Rule $LU$-valid trace we know that for all $\alpha \in \pi$ such that relevant($\alpha$) = $H$, this holds: HPFR $M; H \rightsquigarrow M'$.

We can expand HPFR by Rule $LU$-Monitor Step and get:

HPMR: $(\sigma_c, H', \sigma_f) \in \rightsquigarrow$

for a heap $H' \subseteq H$

By HPM $M \approx_\beta M$ for initial states.

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

By the first clause of Definition 8 ($M \text{RM}$) we know that

HPMRBI: $(\sigma_c, H, \_) \in \rightsquigarrow \iff (\sigma_c, H, \_) \in \rightsquigarrow$

By HPMRBI with HPMR we know that

HPMRTC: $(\sigma_c, H', \sigma_f) \in \rightsquigarrow$

However, by HPRMT1 and Rule $LP$-valid trace we know that

HPNR: $M; H \not\rightsquigarrow$

so we get

HPCON: $\not\exists (\sigma_c, H', \sigma_f) \in \rightsquigarrow$

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

\[ \Box \]
12.4 Proof of Lemma 1 (Compiled code steps imply existence of source steps)

Proof. The proof proceeds by induction on $\alpha!\Rightarrow$.

Base case: $\alpha!\Rightarrow$

By Rule ELP-single we need to prove the silent steps and the $\alpha!$ action.

The proof proceeds by analysis of the target reductions.

- Rule ELP-sequence In this case we do not need to pick and the thesis holds by Rule ELU-sequence.
- Rule ELP-step In this case we do not need to pick and the thesis holds by Rule ELU-step.
- Rule ELP-if-true We have: $H \triangleright [e]_{LP} \quad \overset{\beta}{\Rightarrow} \quad 0$
  We apply Lemma 9 (Compiled code expression steps implies existence of source expression steps) and obtain a $v \approx_{\beta} 0$
  By definition we have $0 \approx_{\beta} 0$ and $\text{true} \approx_{\beta} 0$, we pick the second.
  So we have $H \triangleright e \quad \overset{\beta}{\Rightarrow} \quad \text{true}$
  We can now apply Rule ELU-if-true and this case follows.
- Rule ELP-if-false This is analogous to the case above.
- Rule ELP-assign-top Analogous to the case above.
- Rule ELP-assign-k This is analogous to the case above but for $v = \ell \approx_{\beta} (n, k)$.
- Rule ELP-letin This follows by Lemma 9 and by Rule ELU-letin.
- Rule ELP-new This follows by Lemma 9 and by Rule ELU-alloc.
- Rule ELP-hide By analysis 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 ELP-call-internal This follows by Lemma 9 and by Rule ELU-call-internal.
- Rule ELP-ret-internal In this case we do not need to pick and the thesis holds by Rule ELU-ret-internal.

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).

\[
\forall \quad \text{if } H \triangleright [e]_{LP}^{LU} \rightarrow v \\
\text{and if } \{\rho\} = \{\rho \mid \rho \approx_\beta \rho\} \\
v \approx_\beta v \\
H \approx_\beta H \\
\text{then } \exists \rho_j \in \{\rho\}. H \triangleright e_{\rho_j} \rightarrow v
\]

Proof. This proceeds by structural induction on \(e\).

Base case: true  
This follows from Rule (\([\cdot]_{LP}^{LU}-\text{True}\)).

false  
This follows from Rule (\([\cdot]_{LP}^{LU}-\text{False}\)).

\(n \in \mathbb{N}\)  
This follows from Rule (\([\cdot]_{LP}^{LU}-\text{nat}\)).

\(x\)  
This follows from the relation of the substitutions and the totality of \(\approx_\beta\) and Rule (\([\cdot]_{LP}^{LU}-\text{Var}\)).

\(\langle v, v' \rangle\)  
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  
\(n \approx_\beta n\)

IHV2  
\(n' \approx_\beta n'\)

By IH we get

IHTE1  
\(H \triangleright [e]_{LP}^{LU} \rightarrow n\)

IHTE2  
\(H \triangleright [e']_{LP}^{LU} \rightarrow n'\)

IHSE1  
\(H \triangleright e_{\rho_j} \rightarrow n\)

IHSE2  
\(H \triangleright e'_{\rho_j} \rightarrow n'\)

By Rule (\([\cdot]_{LP}^{LU}-\text{op}\)) we have that \([e \oplus e']_{LP}^{LU} = [e]_{LP}^{LU} \oplus [e']_{LP}^{LU}\).

By Rule \(ELP^\text{-op}\) with IHTE1 and IHTE2 we have that \(H \triangleright [e]_{LP}^{LU} \oplus [e']_{LP}^{LU} \rightarrow n''\)
where IHVT \(n'' = n \oplus n'\)

By Rule \(ELU^\text{-op}\) with IHSE1 and IHSE2 we have that \(H \triangleright e \oplus e' \rightarrow n''\)
if \(n'' = n \oplus n'\)

This follows from IHVT and IHV1 and IHV2.
e \otimes e' As above, this follows from IH and Rule ([L_P^L_L^u\text{-cmp}]) and Rule EL^u\text{-comp}.

\langle e, e' \rangle As above, this follows from IH and Rule ([L_P^L_L^u\text{-Pair}]).

e.1 As above, this follows from IH and Rule ([L_P^L_L^u\text{-P1}]) and Rule EL^u\text{-p1}.

e.2 Analogous to the case above.

!e As above, this follows from IH and Rule ([L_P^L_L^u\text{-Deref}]) and Rule EL^u\text{-dereference} but with the hypothesis that e evaluates to a v related to \langle n, v \rangle.

\square

\fill

12.5 Proof of Theorem 4 \((\langle \cdot \rangle_{L_P}^{L_L^u} \text{ is correct})\)

Proof. HP1 \(\Omega_0 (A [\|C]\|_{L_P}^{L_L^u}) \overset{\pi}{\Rightarrow} \Omega\)

HPF \(\Omega \overset{\pi}{\Rightarrow} \Omega'\)

HPN \(I = \text{names}(A)\)

HPT \(\pi \equiv \alpha' \cdot \alpha?\)

HPL \(\ell_i; \ell_{\text{glob}} \notin \beta\)

THE \(\exists A \in \langle I, \pi \rangle_{L_P}^{L_L^u}\)

TH1 \(\Omega_0 (A | C) \overset{\pi}{\Rightarrow} \Omega\)

THA \(\pi \simeq_{\beta} \bar{\pi}\)

THS \(\Omega \simeq_{\beta} \Omega\)

THC \(\Omega.\ell_i = \|\pi\| + 1\)

The proof proceeds by induction on \(\bar{\alpha}'\).

Base case: We perform a case analysis on \(\alpha?\)

call f v H?

Given

HP1 \(\Omega_0 (A [\|C]\|_{L_P}^{L_L^u}) \overset{\text{call f v H?}}{\Rightarrow} \Omega\)

We need to show that

THE \(\exists A \in \langle I, \pi \rangle_{L_P}^{L_L^u}\)

TH1 \(\Omega_0 (A | C) \overset{\pi}{\Rightarrow} \Omega\)

THA \(\text{call f v H?} \simeq_{\beta} \text{call f v H?}\)

THS \(\Omega \simeq_{\beta} \Omega\)

THC \(\Omega.\ell_i = \|\pi\| + 1\)

By Rule \((\langle \cdot \rangle_{L_P}^{L_L^u}\text{-call})\) the back-translated context executes this code inside main:
if ℓ_i == n then
    incrementCounter()
    let x1 = new v_1 in register(⟨x1, n_1⟩)
    ...
    let x_j = new v_j in register(⟨x_j, n_j⟩)
    call f v
else skip
As H_prev is ∅, no updates are added.

Given that ℓ_i is initialised to 1 in Rule (⟨⟨·⟩⟩_{L^P}skel), this code is executed and it generates action call f v H? where H=ℓ_1 → v_1; · · · ; ℓ_n → v_n for all n_i ∈ dom(H) such that ℓ_i ≈ β ⟨n_i, _⟩ and:

HPR H ≈_β H

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

By Rule Related states — Secure, THS holds too.

Execution of incrementCounter() satisfies THC.

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

Inductive case:

We know that (eliding conditions HP that are trivially satisfied):

IH-HP1 Ω_0 (A [ [L^P_{L^P}] ] ) ⤐→ Ω′ ⤐→ Ω′′ ⤐→ Ω

And we need to prove:

IH-TH1 Ω_0 (⟨⟨I, π⟩⟩_{L^P} [C] ) ⤐→ Ω′ ⤐→ Ω′′ ⤐→ Ω

IH-THA πa!a? ≈_β πa!a?

IH-THS Ω ≈_β Ω

And the inductive HP is (for ∅ ⊆ β'):

IH-HP1 Ω_0 (A [ [L^P_{L^P}] ] ) ⤐→ Ω′
IH-TH1 Ω_0 (⟨⟨I, π⟩⟩_{L^P} [C] ) ⤐→ Ω′
IH-THA π ≈_β π
IH-THS Ω′ ≈_β Ω′
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 α!, and show that the back-translated code performs α!.

By IH we have that the existing code is generated by Rule (⟨⟨·⟩⟩L^P-listact-i): ⟨⟨π, n, H_{pre}, ak, f⟩⟩_{L^P}.

The next action α! produces code according to:

HPF ⟨⟨α!, n, H_{pre}, ak, f⟩⟩_{L^P}.

By Rule (⟨⟨·⟩⟩L^P-join), code of this action is the first if statement executed.

call f v H! By Rule (⟨⟨·⟩⟩L^P-callback-loc) this code is placed at function f so it is executed when compiled code jumps there

if !ℓ_{i} == n then
  incrementCounter()
  let l1 = e_{i} in register((l1, n_{i}))
  ...  
  let lj = e_{j} in register((lj, n_{j}))
else skip

By IH we have that ℓ_{i} ↦→ n_{i}, so we get

IHL ℓ_{i} ↦→ n_{i} + 1

By Definition 15 (Reachable) we have for i ∈ 1..j that a reachable location n_{i} ∈ dom(H) has a related counterpart in ℓ_{i} ∈ dom(H) such that H ⊢ e_{i} ↦→ ℓ_{i}.

By Lemma 10 (L^P attacker always has access to all capabilities) we know all capabilities to access any 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 f = f′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 (⟨⟨·⟩⟩L^P-ret-loc)

if !ℓ_{i} == n then
  incrementCounter()
  let l1 = e_{i} in register((l1, n_{i}))
  ...  
  let lj = e_{j} in register((lj, n_{j}))
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: $\Omega'' \approx_{\beta''} \Omega''$

The next action $\alpha?$ produces code according to:

IHFA 1: $\langle\langle \alpha?, n + 1, H'_\text{pre}, ak', \tau \rangle\rangle_{L^P}$

We perform a case analysis on $\alpha?$ and show that the back-translated code performs $\alpha?$:

- **ret H?** By Rule ($\langle\langle \cdot \rangle\rangle_{L^P}$-ret back), after $n$ actions, we have from IHF1 that $\tau = \ell' \ell''$ and inside function $f'$ there is this code:
  
  ```
  if !\ell_i \implies \lnot n then
    let x_1 = new v_1 in register((x_1, n_1))
    ...
    let x_j = new v_j in register((x_j, n_j))
    update(m_1, u_1)
    ...
    update(m_l, u_l)
  else skip
  ```

  By IHL, $\ell_i \implies n + 1$, so the `if` gets executed.

  By definition, for all $n \in \text{dom}(H)$ we have that $n \in H_n$ or $n \in H_c$ (from the case definition).

  By Lemma 10 (L^P attacker always has access to all capabilities) we know all capabilities to access any $n$ are in $ak$.

  We induce on the size of $H$; the base case is trivial and the inductive case follows from IH and the following:

- **H_n:** and $n$ is newly allocated.
  
  In this case when we execute
  
  ```
  C; H'; let x_1 = new v_1 in register((x_1, n_1)) \xrightarrow{\ell'} C; H'; \ell'' \xrightarrow{\ell''} \langle\langle v_1 \rangle\rangle_{L^P} \xrightarrow{\ell''} \text{register}((\ell'', n_1))
  ```
  
  and we create $\beta''$ by adding $\ell'', n, \eta'$ to $\beta'$.

  By Lemma 2 (register($\ell, n$) does not add duplicates for $n$) we have that:
  
  ```
  C; H'; \ell'' \xrightarrow{\ell''} \langle\langle v_1 \rangle\rangle_{L^P} \xrightarrow{\ell''} \text{register}((\ell'', n_1)) \xrightarrow{\ell''} C; H'; \ell'' \xrightarrow{\ell''} \langle\langle v_1 \rangle\rangle_{L^P} \xrightarrow{\ell''} \text{skip}
  ```
  
  and we can lookup $\ell''$ via $n$.

- **H_c:** and $n$ is already allocated.
  
  In this case
  
  ```
  C; H'; update(m_1, u_1) \xrightarrow{\ell''} C; H'' \xrightarrow{\ell''} \text{skip}
  ```

  By Lemma 3 (update($n, v$) never gets stuck) we know that $H'' = H''[\ell'' \xrightarrow{\ell''} \text{skip}]$ and $\ell''$ such that $(\ell'', m_1, \eta'') \in \beta'$.
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 $HPRH \approx_{\beta''} H$.

As no other if inside $f$ is executed, eventually we hit its return statement, which by Rule $(\langle\langle\cdot\rangle\rangle_{LU}^{LP}$-join) and Rule $(\langle\langle\cdot\rangle\rangle_{LU}^{LP}$-fun) is $\text{incrementCounter(); return;})$.

Execution of $\text{incrementCounter()}$ satisfies THC.

So we have $\Omega'' \xrightarrow{\text{ret}} H \Rightarrow \Omega$ (by Lemma 11) and with HPRH.

call $f$ v $H$? 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.

\[\Box\]

\[\Box\]

**Lemma 10** ($L^r$ attacker always has access to all capabilities).

\[\forall v, \beta. \langle\langle v \rangle\rangle_{LU}^{LP} \approx_{\beta} v\]

**Proof.** Trivial case analysis on Rules $(\langle\langle\cdot\rangle\rangle_{LU}^{LP}$-ret-loc) to $(\langle\langle\cdot\rangle\rangle_{LU}^{LP}$-callback-loc).

\[\Box\]

\[\Box\]

**Lemma 11** (Backtranslated values are related).

\[\forall v, \beta. \langle\langle v \rangle\rangle_{LU}^{LP} \approx_{\beta} v\]

**Proof.** Trivial analysis of Section 4.2.1.

\[\Box\]
12.6 Proof of Theorem 5 (Typability Implies Robust Safety in $L^\tau$)

Proof. HP1 $\vdash C : UN$
HP2 $C \vdash M$
TH $M \vdash C : rs$

We expand TH: $\forall A, \pi. M \vdash A : attacker$ and $\vdash A [C] : whole$ if HPR $O_\emptyset (A [C]) \xrightarrow{\pi} _$
then THM $M \vdash \pi$

By definition of relevant($\pi$) and by Rule $L^\tau$-valid trace we get a $\overline{H}$ to induce on.

The base case holds by Rule $L^\tau$-Monitor Step Trace Base.
In the inductive case we are considering $\overline{H} \cdot H$ and the IH covers the first part of the trace.

By Lemma 12 ($L^\tau$-$\alpha$ reductions respect heap typing), given that the state generating the action is $C, H \triangleright \Pi \rho$ we know that, HPH $\vdash$ mon-care($H, \Delta$) : $\Delta$

By Rule $L^\tau$-valid trace and by Rule $L^\tau$-Monitor Step we need to show that $\vdash H : \Delta$

This follows by HPH.

We thus need to prove that the initial steps are related heaps are secure.
By Rule $L^\tau$-Plug we need to show that the heaps constituting the initial heap - both $H$ and $H_0$ - are well typed.

The latter, $\vdash H_0 : \Delta$, holds by Rule $L^\tau$-Plug.

The former holds by definition of the attacker: Rules TUL$^\tau$-base and TUL$^\tau$-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 $L^\tau$-Plug.

Lemma 12 ($L^\tau$-$\alpha$ reductions respect heap typing).

\[
\text{if } C \equiv \Delta; \cdots \\
\vdash \text{mon-care}(H, \Delta) : \Delta \\
C, H \triangleright \Pi \rho \xrightarrow{\pi} C', H' \triangleright \Pi' \rho'
\]
then $\vdash \text{mon-care}(H', \Delta) : \Delta$

Proof. The proof proceeds by induction on $\pi$. 

72
Base case This trivially holds by HP.

Inductive case This holds by IH plus a case analysis on the last action:

- call f v? This holds by Lemma 17 (Lτ-? actions respect heap typing).
- call f v! This holds by Lemma 18 (Lτ-! actions respect heap typing).
- ret ! This holds by Lemma 18 (Lτ-! actions respect heap typing).
- ret ? This holds by Lemma 17 (Lτ-? actions respect heap typing).

Lemma 13 (Lτ An attacker only reaches UN locations).

∀
if ℓ′ → v : UN ∈ H
then ♠
H ⊢ e ↔ ℓ′
ℓ′ → v : τ ∈ H
τ ≠ UN

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 TLτ-assign prevents it.

Lemma 14 (Lτ attacker reduction respects heap typing).

if C ≡ Δ; · · ·
C Γatt ↠ Nil → Nil′
C, H ⊢ Πρ ↠ C, H′ ⊢ Πρ′
then mon-care(H, Δ) = mon-care(H′, Δ)
Proof. Trivial induction on the derivation of $\Pi$, which is typed with $\vdash_{UN}$ and by Lemma 13 (L$^\tau$ An attacker only reaches UN locations) has no access to locations in $\Delta$ or with a type $\tau \vdash \circ$. □

Lemma 15 (L$^\tau$ typed reduction respects heap typing).

if $C \equiv \Delta; \ldots$

$C, \Gamma \vdash s$

$C, \Gamma \vdash s'$

$\vdash \text{mon-care}(H, \Delta) : \Delta$

$C, H\triangleright s \trianglerightrightarrow C', H'\triangleright s'\rho'$

then $\vdash \text{mon-care}(H', \Delta) : \Delta$

Proof. This is done by induction on the derivation of the reducing statement. There, the only non-trivial cases are:

Rule $T_L^\tau$-new By IH we have that

$H \triangleright e\rho \iff v$

So

$C; H \triangleright \text{let } x = \text{new}_{\tau} e \text{ in } s \trianglerightrightarrow C; H \triangleright v : \tau \triangleright s[\ell / x]$

By IH we need to prove that $\vdash \text{mon-care}(\ell \mapsto v : \tau, \Delta) : \Delta$

As $\ell \notin \text{dom}(\Delta)$, by Rule $L^\tau$-Heap-ok-i this case holds.

Rule $T_L^\tau$-assign By IH we have (HPH) $H \triangleright e \iff v$

such that $\ell : \text{Ref } \tau$ and $v : \tau$.

So

$C; H \triangleright x := e\rho \trianglerightrightarrow C; H' \triangleright \text{skip}$

where $[x / \ell] \in \rho$ and $H = H_1; \ell \mapsto \ell' \mapsto v : \tau; H_2$

$H' = H_1; \ell \mapsto v' : \tau; H_2$

There are two cases

$\ell \in \text{dom}(\Delta)$ By Rule $L^\tau$-Heap-ok-i we need to prove that $\ell : \text{Ref } \tau \in \Delta$.

This holds by HPH and Rule $L^\tau$-Initial State, as the initial state ensures that location $\ell$ in the heap has the same type as in $\Delta$.

$\ell \notin \text{dom}(\Delta)$ This case is trivial as for allocation.
Rule **TL-coercion** We have that $C, \Gamma \vdash e : \tau$ and HPT $\tau \vdash \circ$.

By IH $H \triangleright e \leftrightarrow v$ such that $\vdash \text{mon-care}(H', \Delta) : \Delta$.

By HPT we get that $\text{mon-care}(H) = \text{mon-care}(H')$ as by Rule $L\tau$-Secure.

Heap function $\text{mon-care}(\cdot)$ only considers locations whose type is $\tau \neq \circ$, so none affected by $e$.

So this case by IH.

**Rule TL-endorse** By Rule $EL\tau$-endorse we have that $H \triangleright e \leftrightarrow v$ and that $C, H \triangleright \text{endorse } x = e$ as $\varphi$ in $s \leftrightarrow C, H \triangleright s[v / x]$.

So this holds by IH.

\[\square\]

\[\blacklozenge\]

**Lemma 16** ($L\tau$ any non-cross reduction respects heap typing).

\[
\text{if } C \equiv \Delta; \cdots
\]

\[
\vdash \text{mon-care}(H, \Delta) : \Delta
\]

\[
C, H \triangleright \Pi \rho \xrightarrow{\lambda} C', H' \triangleright \Pi' \rho'
\]

then $\vdash \text{mon-care}(H', \Delta) : \Delta$

\[\text{Proof. By induction on the reductions and by application of Rule } EL\tau\text{-par. The base case follows by the assumptions directly. In the inductive case we have the following:}
\]

\[
C, H \triangleright \Pi \rho \xrightarrow{\lambda} C'', H'' \triangleright \Pi'' \rho'' \xrightarrow{\lambda} C', H' \triangleright \Pi' \rho'
\]

This has 2 sub-cases, if the reduction is in an attacker function or not.

$C \vdash_{\text{att}} \Pi'' \rightarrow \Pi$: this follows by induction on $\Pi''$ and from IH and Lemma 14 ($L\tau$ attacker reduction respects heap typing).

$C \nvdash_{\text{att}} \Pi'' \rightarrow \Pi$: In this case we induce on $\Pi''$.

The base case is trivial.

The inductive case is $(s)_{\tau} \parallel \Pi$, which follows from IH and Lemma 15 ($L\tau$ typed reduction respects heap typing).

\[\square\]

\[\blacklozenge\]

75
Lemma 17 ($L^\tau$-? actions respect heap typing).

If $C \equiv \Delta; \cdots$

$$C, H \triangleright \Pi \rho \rightarrow \gamma \Rightarrow C', H' \triangleright v'$$

then $\text{mon-care}(H, \Delta) = \text{mon-care}(H', \Delta)$

Proof. By Lemma 16 ($L^\tau$ any non-cross reduction respects heap typing), and a simple case analysis on $\alpha$? (which does not modify the heap). □

Lemma 18 ($L^\tau$-! actions respect heap typing).

If $C \equiv \Delta; \cdots$

$$C, H \triangleright \Pi \rho \rightarrow \gamma \Rightarrow C', H' \triangleright v'$$

$\vdash \text{mon-care}(H, \Delta) : \Delta$

then $\vdash \text{mon-care}(H', \Delta) : \Delta$

Proof. By Lemma 16 ($L^\tau$ 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 $[\_]^{L^\tau}_{L^\pi}$ is CC)

Proof. By definition initial states have related components, related heaps and well-typed, related starting processes, for $\beta_0 = (\text{dom}(\Delta), \text{dom}(H_0), H_0, \eta)$ so we have:

$$\text{HRS } \Omega_0(C) \approx_{\beta_0} \Omega_0([C]^{L^\tau}_{L^\pi}).$$

As the languages have no notion of internal nondeterminism we can apply Lemma 20 (Generalised compiler correctness for $[\_]^{L^\tau}_{L^\pi}$) with HRS to conclude. □

Lemma 19 (Expressions compiled with $[\_]^{L^\tau}_{L^\pi}$ are related).

$$\forall C \equiv \Delta; \cdots$$

If $H \approx_{\beta} H$

$$H \triangleright e \rho \rightarrow \nu$$

Then $H \triangleright [e]^{L^\tau}_{L^\pi} \vdash [\nu]^{L^\tau}_{L^\pi}$

76
Proof. The proof is analogous to that of Lemma 7 (Expressions compiled with \([L^T]\) are related) as the compilers perform the same steps and expression reductions are atomic. \(\square\)

Lemma 20 (Generalised compiler correctness for \([L^T]\)).

\[\forall \ldots \exists \beta' \text{ if } C; \Gamma \vdash \Pi, \]
\[\vdash C : \text{whole} \]
\[C = \Delta; \bar{F}; \bar{I} \]
\[\langle C \rangle_{L^T} = H_0; \bar{F}; \bar{I} = C \]
\[C, H \triangleright \Pi \approx \beta C, H \triangleright \langle C; \Gamma \vdash \Pi \rangle_{L^T} \]
\[C, H \triangleright \Pi; \rho \Rightarrow C, H' \triangleright \Pi'; \rho' \]
\[
\text{then } \begin{align*}
C, H & \triangleright \langle C; \Gamma \vdash \Pi \rangle_{L^T} = C, H' \triangleright \langle C; \Gamma \vdash \Pi \rangle_{L^T} \Rightarrow C, H \triangleright \langle C; \Gamma \vdash \Pi \rangle_{L^T} \\
C, H & \triangleright \Pi; \rho' \Rightarrow \beta' C, H \triangleright \langle C; \Gamma \vdash \Pi \rangle_{L^T} \Rightarrow C, H' \triangleright \langle C; \Gamma \vdash \Pi \rangle_{L^T} \\
\beta & \subseteq \beta'
\end{align*}
\]

Proof. This proof proceeds by induction on the typing of \(\Pi\) and then of \(\pi\).

Base Case skip Trivial by Rule \((L^T\text{-Skip})\).

Inductive Case

In this case we proceed by induction on the typing of \(s\)

Inductive Cases Rule TL\text{-new} There are 2 cases, they are analogous.

\(\tau = UN\) By HP

\[\Gamma \vdash e : \tau \]
\[H \triangleright e \leftrightarrow v \]
\[C, H \triangleright \text{let } x = \text{new}_\tau e \text{ in } s\rho \quad \rightarrow C, H, \ell \mapsto v : \tau \triangleright s[\ell / x]_\rho \]

By Lemma 19 we have:

IHR1 \[H \triangleright \langle \Gamma \vdash e : \tau \rangle_{L^T} \quad \rightarrow \quad \langle \Gamma \vdash v : \tau \rangle_{L^T} \]

By Rule \((L^T\text{-New})\) we get

let xo = new \[\langle \Gamma \vdash e : \tau \rangle_{L^T} \]

in let x = \langle xo, 0 \rangle

in \[\langle C, \Gamma ; x : \text{Ref } \tau \vdash s \rangle_{L^T} \]

So:

\[C, H \triangleright \text{let } xo = \text{new} \langle \Gamma \vdash e : \tau \rangle_{L^T} \]

in let x = \langle xo, 0 \rangle

in \[\langle C, \Gamma ; x : \text{Ref } \tau \vdash s \rangle_{L^T} \]

77
\[ \xrightarrow{\bar{\tau}} C, H; n \mapsto [\Gamma \vdash \tau]_{L^T} : \bot \; \text{let} \; x = \langle n, 0 \rangle \]

By Rule \( C, \Gamma; x \vdash \text{Ref } \tau \vdash s \)\( ]_{L^T} \)

\[ \xrightarrow{\bar{\tau}} C, H; n \mapsto [\Gamma \vdash \tau]_{L^T} : \bot \; \text{let} \; [C, \Gamma; x \vdash \text{Ref } \tau \vdash s]_{L^T} \langle \langle n, 0 \rangle / x \rangle \]

For \( \beta' = \beta \cup (\ell, n, \bot) \), this case holds.

**else** The other case holds follows the same reasoning but for \( \beta' = \beta \cup (\ell, n, k) \) and for \( H' = H; n \mapsto [C, \Gamma \vdash \tau]_{L^T} : k; k \).

**Rule TL^T-sequence** By HP

- \( \Gamma \vdash s; \Gamma \vdash s' \)
- \( C, H \vdash s \rho \Rightarrow C', H' \vdash s'' \rho'' \)

There are two cases

- \( s'' = \text{skip} \)

**Rule EL^U-sequence**

- \( C', H' \vdash \text{skip} \rho''; s' \rho \xrightarrow{\bar{\tau}} C', H' \vdash s' \rho \)

By IH

\[ C, H \vdash [\Gamma \vdash s]_{L^T} \[ [\rho]_{L^T} \] \Rightarrow C', H' \vdash [\Gamma \vdash \text{skip}]_{L^T} [\rho']_{L^T} \]

By Rule (\( \parallel \)\( L^T \)-Seq)

\[ [C, \Gamma \vdash s]_{L^T} ; [C, \Gamma \vdash s']_{L^T} \]

So

\[ C, H \vdash [C, \Gamma \vdash s]_{L^T} \[ [\rho]_{L^T} \] \Rightarrow [C', \Gamma \vdash s]_{L^T} \[ [\rho']_{L^T} \] \]

\[ \xrightarrow{\bar{\tau}} C', H' \vdash [C, \Gamma \vdash s']_{L^T} [\rho]_{L^T} \]

At this stage we apply IH and the case holds.

**else** By Rule \( EL^U \)-step we have

- \( C, H \vdash s; s' \Rightarrow C', H' \vdash s''; s' \)

This case follows by IH and HPs.

**Rule TL^T-function-call** Analogous to the cases above.

**Rule TL^T-letin** Analogous to the cases above.

**Rule TL^T-assign** Analogous to the cases above.

**Rule TL^T-if** Analogous to the cases above.

**Rule TL^T-fork** Analogous to the cases above.

**Rule TL^T-coercion** By Rule (\( \parallel \)\( L^T \)-Coerce), this follows from IH directly.

**Rule TL^T-endorse** This has a number of trivial cases based on Rule (\( \parallel \)\( L^T \)-Endorse) that are analogous to the ones above.

\[ \square \]

\[ \diamondsuit \]

78
12.8 Proof of Theorem 7 (Compiler $[\cdot]^{\text{L}^{\pi}}_{\text{L}^{\tau}}$ is RSC)

Proof. Given:

HP1: $M \vdash C : rs$

HPM: $M \approx_{\beta} M$

We need to prove:

TP1: $M \vdash [\cdot]^{\text{L}^{\pi}}_{\text{L}^{\tau}} : rs$

We unfold the definitions of $rs$ and obtain:

$\forall A. M \vdash A : \text{attacker}, \vdash A [[C]_{\text{L}^{\tau}}] : \text{whole}$

HPE1: if $\Omega_{\beta}(A[[C]_{\text{L}^{\tau}}]) \Rightarrow_\pi \Rightarrow_\pi$ then $M \vdash \text{relevant}(\pi)$

$\forall A. M \vdash A : \text{attacker}, \vdash A [[C]_{\text{L}^{\tau}}] : \text{whole}$

THE1: if $\text{HPT} \Omega_{\beta}(A[[C]_{\text{L}^{\tau}}]) \Rightarrow_\pi \Rightarrow_\pi$ then $\text{THE1} M \vdash \text{relevant}(\pi)$

By definition of the compiler we have that

HPISR: $\Omega_{\beta}(A[[C]_{\text{L}^{\tau}}]) \Rightarrow_\pi \Omega_{\beta}(A[[C]_{\text{L}^{\tau}}])$

for $\beta = \text{dom}(\Delta), H_{0}$ such that $M = ([\{\sigma\}, \sim, \sigma_{0}, \Delta, \sigma_{c}]$ and $M = ([\{\sigma\}, \sim, \sigma_{0}, H_{0}, \sigma_{c}]$

By relevant(\pi) and Rule $L^{\pi}$-valid trace we get a $H$ to induce on.

Base case: this holds by Rule $L^{\pi}$-Monitor Step Trace Base.

Inductive case: By Rule $L^{\pi}$-Monitor Step Trace, $M; H \rightsquigarrow M'$ holds by IH, we need to prove $M;' ; H \rightsquigarrow M'$.

By Rule $L^{\pi}$-Monitor Step we need to prove that $\text{THMR}: \exists \sigma'. (\sigma, \text{mon-care}(H, H_{0}), \sigma') \in \sim$.

By HPISR and with applications of Lemmas 22 and 23 we know that states are always related with $\approx_{\beta}$ during reduction.

So by Lemma 24 ($\approx_{\beta}$ implies relatedness of the high heaps) we know that $\text{HPHH mon-care}(H, \Delta) \approx_{\beta} \text{mon-care}(H, H_{0})$, for $H, H$ being the last heaps in the reduction.

By HPM and Rule Monitor relation we have $\beta_{0}, \Delta \vdash M$.

By this and Rule OkMon we have that $\text{HPHR \forall mon-care}(H, \Delta) \approx_{\beta} \text{mon-care}(H, H_{0})$.

if $\vdash H : \Delta$ then $\exists \sigma'. (\sigma, \text{mon-care}(H, H_{0}), \sigma') \in \sim$ so by HPHH we can instantiate this with $H$ and $H$.

By Theorem 5 (Typability Implies Robust Safety in $L^{\pi}$) applied to HPE1, as $[[\cdot]^{\text{L}^{\tau}}_{\text{L}^{\tau}}$ operates on well-typed components, we know that $\text{HPMR}: M \vdash \text{relevant}(\pi)$ for all $\pi$.

So by Rule $L^{\tau}$-Monitor Step with HPMT we get $\text{HPHD} \vdash H : \Delta$ for the $H$ above.

By HPHD with HPHR we get $\text{THMR} \exists \sigma'. (\sigma, \text{mon-care}(H, H_{0}), \sigma') \in \sim$, so this case holds.

\[\square\]
Lemma 21 ($\equiv_{\beta}$ implies relatedness of the high heaps).

If $\Omega = \Delta; F; \Gamma; H \triangleright \Pi$

$$\Omega = H_0; F; [F]^L_T; \Gamma; H \triangleright \Pi$$

$$\Omega \equiv_{\beta} \Omega$$

then $\text{mon-care}(H, \Delta) \approx_{\beta} \text{mon-care}(H, H_0)$

Proof. By point 2a in Rule Related states – Secure.

Lemma 22 ($L^T$-compiled actions preserve $\equiv_{\beta}$).

$$\forall \ldots$$

If $C, H \triangleright \Pi \rho \overset{\lambda}{\rightarrow} C, H' \triangleright \Pi' \rho'$

$$C, H \triangleright [C; \Gamma \triangleright \Pi]^L_T[\rho]^L_T \overset{\lambda}{\rightarrow} C, H' \triangleright [C; \Gamma \triangleright \Pi]^L_T[\rho']^L_T$$

$$C, H \triangleright \Pi \rho \equiv_{\beta} C, H \triangleright [C; \Gamma \triangleright \Pi]^L_T[\rho]$$

$$C, \Gamma \triangleright \Pi$$

then $C, H' \triangleright \Pi' \rho' \equiv_{\beta} C, H' \triangleright [C; \Gamma \triangleright \Pi]^L_T[\rho']^L_T$

Proof. Trivial induction on the derivation of $\Pi$, analogous to Lemma 20 (Generalised compiler correctness for $[\cdot]^L_T$).

Rule $T_L^T$-new There are 2 cases, they are analogous.

$\tau = \text{UN}$ By HP

$$\Gamma \vdash e : \tau$$

$H \triangleright e \mapsto v$

$$C, H \triangleright \text{let } x = \text{new}_\tau e \triangleright C, H; \ell \mapsto v : \tau \triangleright s[\ell / x] \rho$$

By Lemma 19 (Expressions compiled with $[\cdot]^L_T$ are related) we have:

IHR1 $H \triangleright [\Gamma \vdash e : \tau]^L_T[\rho]^L_T \mapsto [\Gamma \vdash v : \tau]^L_T$

By Rule ($[\cdot]^L_T$-New) we get

let $xo = \text{new} [\Gamma \vdash e : \tau]^L_T$

in let $x = \langle xo, 0 \rangle$

in $[C, \Gamma ; x : \text{Ref } \tau \vdash s]^L_T$
So:

\[ C, H \triangleright \text{let } xo = \text{new } \llbracket \Gamma \vdash e : \tau \rrbracket_{L^\gamma} \]
\[ \text{in let } x = (xo, 0) \]
\[ \text{in } \llbracket C, \Gamma ; x : \text{Ref } \tau \vdash s \rrbracket_{L^\gamma} \]
\[ \downarrow \rightarrow \quad \text{in } \llbracket C, \Gamma ; x : \text{Ref } \tau \vdash s \rrbracket_{L^\gamma} \]
\[ \downarrow \rightarrow \quad \text{let } x = \langle n, 0 \rangle \]
\[ \text{in } \llbracket C, \Gamma ; x : \text{Ref } \tau \vdash s \rrbracket_{L^\gamma} \]

For \( \beta' = \beta \), this case holds.

**else** The other case holds follows the same reasoning but

for \( \beta' = \beta \cup (\ell, n, k) \) and for \( H' = H; n \mapsto \llbracket C, \Gamma ; v : \tau \rrbracket_{L^\gamma} : k; k \).

We need to show that this preserves Rule Related states Secure, specifically it preserves point \((2a): \ell \approx_{\beta} (n, k) \) and \( \ell \mapsto v : \tau \in H \) and \( v \approx_{\beta} v \)

These follow all from the observation above and by Lemma 19 (Expressions compiled with \( \llbracket L \rrbracket_{L^\gamma} \), are related).

\[ \square \]

---

**Lemma 23** (LP Attacker actions preserve \( \cong \)).

\[ \forall... \]
\[ \text{if } C, H \triangleright \Pi \rho \overset{\lambda}{\rightarrow} C, H' \triangleright \Pi' \rho' \]
\[ C, H \triangleright \Pi \rho \overset{\lambda}{\rightarrow} C, H' \triangleright \Pi' \rho' \]
\[ C, H \triangleright \Pi \rho \overset{\equiv_{\beta}}{\rightarrow} C, H \triangleright \Pi \rho \]
\[ C \triangleright_{\text{att}} \Pi \rho \overset{\lambda}{\rightarrow} \Pi' \rho' \]
\[ C \triangleright_{\text{att}} \Pi \rho \overset{\lambda}{\rightarrow} \Pi' \rho' \]
\[ \text{then } C, H' \triangleright \Pi' \rho' \overset{\equiv_{\beta}}{\rightarrow} C, H' \triangleright \Pi' \rho' \]

**Proof.** For the source reductions we can use Lemma 16 \( (L^\gamma \text{ any non-cross reduction respects heap typing}) \) to know that \( \text{mon-care}(H) = \text{mon-care}(H') \), so they don’t change the interesting bits of the \( \cong_{\beta} \).

Suppose this does not hold by contradiction, there can be three clauses that do not hold based on Rule Related states Secure:

- violation of \((1): \exists \pi \in \Pi. C \vdash \pi : \text{attacker} \text{ and } k \in \text{fv}(\pi) \).

By HP5 this is a contradiction.

81
• violation of (2a): $n \rightarrow v: k \in H$ and $\ell \not\approx\beta (n,k)$ and $\ell \rightarrow v: \tau \in H$ and $\neg (v \approx\beta v)$

To change this value the attacker needs $k$ which contradicts points (1) and (2b).

• violation of (2b): either of these:
  
  - $H, H \not\vdash \text{low-loc}(n')$
    Since Rule $L^\pi$:High Location does not hold, by Lemma 5 this is a contradiction.
  
  - $v = k'$ for $H, H \vdash \text{high-cap}(k')$
    This can follow from another two cases
    * forgery of $k$: an inspection of the semantics rules contradicts this
    * update of a location to $k'$: however $k'$ is not in the code (contradicts point (1)) and by induction on the heap $H$ we have that $k'$ is stored in no other location, so this is also a contradiction.

\[ \square \]

\[ \bullet \]

12.9 Proofs for the Non-Atomic Variant of $L^\pi$ (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:

\[ \Omega \sim_\beta \Omega \]

(Non Atomic State Relation)

\[ \Omega \approx_\beta \Omega \quad \frac{\Omega \sim_\beta \Omega}{\Omega \sim_\beta \Omega} \]

(Non Atomic State Relation -attack)

\[ \Omega = C, H \triangleright \Pi \quad \Omega = C, H \triangleright \Pi \]

\[ \exists \pi \in \Pi. C \not\triangleright \pi : \text{attacker} \]

\[ \pi = (\text{hide } n : s_f)_{\tau_f} \quad \exists f \in \text{dom(} \tau_f \text{)}. f \approx_\beta f \]

\[ \forall \ell, \ell \in \text{dom(} \triangleright \text{secure}(H)) \quad n \rightarrow v; k \in H \quad \ell \not\approx_\beta (n,k) \]

\[ \Omega \sim_\beta \Omega \]

Two states are now related if:

• either they are related by $\approx_\beta$

• or the red process is stuck on a $\text{hide } n$ where $n \rightarrow v; k$ but $\ell \sim (n,k)$ does not hold for a $\ell$ that is secure, and we have that $\ell \sim (n,0)$ (as this was after the new). And the $\text{hide}$ 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 (\([\mathbb{L}^r]_\text{New-nat}\)) as follows:

Proof. \(\mathtt{new} \cdot \) is implemented as defined in Rule (\([\mathbb{L}^r]_\text{New-nat}\)).

\(\tau \neq \mathtt{UN}\) By HP
\[
\Gamma \vdash e : \tau
\]
\[
H \triangleright e \rightarrow v
\]
\[
C, H \triangleright \mathtt{let} \ x = \mathtt{new}_\tau \ e \ \mathtt{in} \ s \rho \rightarrow C, H; \ell \triangleright v : \tau \triangleright s[\ell / x] \rho
\]
By Lemma 19 we have:

IHR1 \(H \triangleright \Gamma \vdash e : \tau\) \(\mathbb{L}^r\) \(\rho\) \(\mathbb{L}^r\) \(\Gamma \vdash v : \tau\) \(\mathbb{L}^r\)

By Rule (\([\mathbb{L}^r]_\text{New-nat}\)) we get

let \(x = \mathtt{new} \ 0 \ \mathtt{in}

let \(x_k = \mathtt{hide} \ x \ \mathtt{in}

let \(x_c = \left[\Delta, \Gamma \vdash e : \tau\right]_{\mathbb{L}^r}\) in

\(x := x_c \ \mathtt{with} \ x_k\);

\([C, \Delta, \Gamma \vdash s]_{\mathbb{L}^r}\)

So:

\[
C, H \triangleright \mathtt{let} \ x = \mathtt{new} \ 0 \ \mathtt{in}

\]

let \(x_k = \mathtt{hide} \ x \ \mathtt{in}

let \(x_c = \left[\Delta, \Gamma \vdash e : \tau\right]_{\mathbb{L}^r}\) in

\(x := x_c \ \mathtt{with} \ x_k\);

\([C, \Delta, \Gamma \vdash s]_{\mathbb{L}^r}\)

\(\rightarrow C, H, n \rightarrow 0 : \bot \triangleright \mathtt{let} \ x_k = \mathtt{hide} \ n \ \mathtt{in}

\]

let \(x_c = \left[\Delta, \Gamma \vdash e : \tau\right]_{\mathbb{L}^r}\) in

\(x := x_c \ \mathtt{with} \ x_k\);

\([C, \Delta, \Gamma \vdash s]_{\mathbb{L}^r}\)

And \(\beta' = \beta \cup (\ell, n, 0)\).

Now there are two cases:

- A concurrent attacker reduction performs \(\mathtt{hide} \ n\), so the state changes.
C, H, n \mapsto 0 : k; k \triangleright let xk = hide n in
let xc = \llbracket \mathcal{C}, \Gamma \vdash e : \tau \rrbracket^L_{\mathcal{L}} \in x := xc with xk;
\llbracket \mathcal{C}, \Delta, \Gamma \vdash s \rrbracket^L_{\mathcal{L}}

At this stage the state is stuck: Rule EL-P-hide does not apply.
Also, we have that this holds by the new \beta^\prime: (\ell \leadsto (n, 0))
And so this does not hold: (\ell \leadsto (n, k))
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 \llbracket \cdot \rrbracket^L_{\mathcal{L}} is CC)

Proof. Analogous to that of Section 12.7.

12.11 Proof of Theorem 9 (Compiler \llbracket \cdot \rrbracket^L_{\mathcal{L}} is RSC)

Proof. Given:
HP1: M \vdash C : rs
HPM: M \cong M
We need to prove:
TP1: M \vdash \llbracket \mathcal{C} \rrbracket^L_{\mathcal{L}} : rs
We unfold the definitions of rs and obtain:
\forall A, M \vdash A : attacker, \vdash A \llbracket \mathcal{C} \rrbracket^L_{\mathcal{L}} : whole
HPE1: if \Omega_0 (A \llbracket \mathcal{C} \rrbracket) \Rightarrow_\pi then M \vdash relevant(\pi)
\forall A, M \vdash A : attacker, \vdash A \llbracket \mathcal{C} \rrbracket^L_{\mathcal{L}} : whole
THE1: if HPRT \Omega_0 (A \llbracket \mathcal{C} \rrbracket^L_{\mathcal{L}}) \Rightarrow_\pi then THE1 M \vdash relevant(\pi)
By definition of the compiler we have that
HPISR: \Omega_0 (A \llbracket \mathcal{C} \rrbracket) \cong_{\varphi} \Omega_0 (A \llbracket \mathcal{C} \rrbracket^L_{\mathcal{L}})
for \varphi = dom(\Delta), H_0 such that M = (\sigma_0, \Delta, \sigma) and M = (\sigma, \Delta, \sigma_0, H_0, \sigma_0)
By relevant(\pi) and Rule \mathcal{L}^L-\text{valid trace} we get a \overline{H} to induce on.

Base case: this holds by Rule \mathcal{L}^L-\text{Monitor Step Trace Base.}
**Inductive case:** By Rule $L^I$-Monitor Step Trace, $M; H \leadsto M''$ holds by IH, we need to prove $M''; H \leadsto M'$. 

By Rule $L^I$-Monitor Step we need to prove that THMR: $\exists \sigma'.(\sigma, \text{mon-care}(H, H_0), \sigma') \in \rightsquigarrow$.

By HPISR and with applications of Lemmas 25 and 26 we know that states are always related with $\equiv_\phi$ during reduction.

So by Lemma 24 ($\approx_\phi$ implies relatedness of the high heaps) we know that HPHH $\text{mon-care}(H, \Delta) \equiv_\phi \text{mon-care}(H, H_0)$, for $H, H'$ being the last heaps in the reduction.

By HPM and Rule Monitor relation (adjusted for $L^I$) we have $\phi_0, \Delta \vdash M$.

By this and Rule Ok Mon (adjusted for $L^I$) we have that HPHR $\forall \text{mon-care}(H, \Delta) \equiv_\phi \text{mon-care}(H, H_0).$ if $\vdash H : \Delta$ then $\exists \sigma'.(\sigma, \text{mon-care}(H, H_0), \sigma') \in \rightsquigarrow$ so by HPHH we can instantiate this with $H$ and $H'$.

By Theorem 5 (Typability Implies Robust Safety in $L^I$) applied to HPE1, as $[\_]'_{L^I}$ operates on well-typed components, we know that HPMR: $M \vdash \text{relevant}(\pi)$ for all $\pi$.

So by Rule $L^I$-Monitor Step with HPMT we get HPHD $\vdash H : \Delta$ for the $H$ above.

By HPHD with HPHR we get THMR $\exists \sigma'.(\sigma, \text{mon-care}(H, H_0), \sigma') \in \rightsquigarrow$, so this case holds.

\[\square\]

\[\blacklozenge\]

**Lemma 24** ($\equiv_\phi$ implies relatedness of the high heaps).

If $\Omega = \Delta; F; F'; I; H \triangleright \Pi$

\[\Omega = H_0; F; [F']^{L^I}; I; E; H \triangleright \Pi\]

$\Omega \approx_\phi \Omega$

then $\text{mon-care}(H, \Delta) \approx_\phi \text{mon-care}(H, H_0)$

*Proof.* By Rule Related states – Secure. \[\square\]

\[\blacklozenge\]

85
Lemma 25 ($L^\tau$-compiled actions preserve $\equiv_\varphi$).

$$\forall \ldots$$

if $C, H \vdash \Pi, \rho \xrightarrow{\lambda} C, H' \vdash \Pi', \rho'$

$$C, H \vdash \llbracket C; \Gamma \vdash \Pi \rrbracket^{L^\tau}_{L'} \xrightarrow{\Delta} C, H' \vdash \llbracket C; \Gamma \vdash \Pi \rrbracket^{L^\tau}_{L'}$$

then $C, H' \vdash \Pi', \rho' \equiv_\varphi C, H' \vdash \llbracket C; \Gamma \vdash \Pi \rrbracket^{L^\tau}_{L'}$

Proof. Trivial induction on the derivation of $\Pi$, analogous to Lemma 20 (Generalised compiler correctness for $[\llbracket \cdot \rrbracket^{L^\tau}_{L'})$ and Lemma 22 ($L^\tau$-compiled actions preserve $\equiv_\beta$).

Lemma 26 ($L^P$ Attacker actions preserve $\equiv$).

$$\forall \ldots$$

if $C, H \vdash \Pi, \rho \xrightarrow{\lambda} C, H' \vdash \Pi', \rho'$

$$C, H \vdash \Pi, \rho \xrightarrow{\lambda} C, H' \vdash \Pi', \rho'$$

then $C, H' \vdash \Pi', \rho' \equiv C, H' \vdash \Pi', \rho' \equiv_\varphi C, H' \vdash \Pi', \rho' \equiv_\varphi$

Proof. For the source reductions we can use Lemma 16 ($L^\tau$ any non-cross reduction respects heap typing) to know that $\text{mon-care}(H) = \text{mon-care}(H')$, so they don’t change the interested bits of the $\equiv_\varphi$.

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 $\ell$ 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 $EL^\ell$-assign-iso and Rule $EL^\ell$-isolate check $C \vdash f : \text{prog}$ and Rule $L^\ell$-Whole ensures that the attacker defines different names from the program, so the attacker can never execute them.

$\square$
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.

```plaintext
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).

```plaintext
clabel testPwd
for i = 0; i < 8; i++ // 8 is the length of the user field in the previous snippet
  add @0 i
  load the memory word stored at address @0 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, $L^U$, nor $L^\tau$, 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 $C_{\text{list}}$ be the program with the List implementation and $C_{\text{tree}}$ 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 $C_{\text{list}}$ and $C_{\text{tree}}$: the code of a RedBlackTree implementation consumes more memory than the code of a List implementation. Thus, a target-level context can distinguish $C_{\text{list}}$ from $C_{\text{tree}}$ by just inspecting the sizes of the code segments. So, in order for $\mathcal{S}_T$ 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 $[3, 8]$.

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 $C_{\text{one}}$ and $C_{\text{two}}$. The only difference between $C_{\text{one}}$ and $C_{\text{two}}$ is that $C_{\text{two}}$ allocates two lists internally; one of these (shadow) is used for internal purposes only.

```java
1 public newList(): List{
2 ell = new List();
3 return ell;
4 }
```

Again, $C_{\text{one}}$ and $C_{\text{two}}$ 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. □
In this example, FAC forces all privately allocated locations to be wrapped in proxies, however RSC does not require this. Our target languages LP and LT support address comparison (addresses are natural numbers in their heaps) but [\llceil_L]u and [\llceil_L]t 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 Cenc (which does the encryption) and Cskip which skips the encryption are equivalent in the source.

```
1 public encryptDB( func : Database → Bitstring) : void {
2     func( this.db );
3     return;
4 }
```

If we compile Cenc and Cskip 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 Cenc with such a func will cause divergence, while calling the compilation of Cskip 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 LU and LT 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 $\text{L}^U$ to $\text{L}^P$

This section sketches a fully abstract compiler from $\text{L}^U$ to $\text{L}^P$.

14.1 Language Extensions to $\text{L}^U$ and $\text{L}^P$

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 $\text{size}$ that returns the amount of locations $\ell$ allocated in the current heap $H$.

In the target language $\text{L}^P$, 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 $\text{isloc}(e)$, $\text{ispair}(e)$ and $\text{eqcap}(e, e)$, respectively.

Finally, compiled code needs private functions for its runtime checks that must not be visible to the context. $\text{L}^P$ does not have this functionality: all functions defined by a component can be called by the context. Now we modify $\text{L}^P$ so that all functions $\mathcal{F}$ 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 $\mathcal{F}$), 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 $\square_{\text{L}^P}$ Compiler

$\square_{\text{L}^P}$ is similar to $\square_{\text{L}^U}$ but with critical differences. We know that fully abstract compilation preserves all source abstractions in the target language. Here, the only abstraction that distinguishes $\text{L}^P$ from $\text{L}^U$ is that locations are abstract in $\text{L}^P$, but concrete natural numbers in $\text{L}^U$. 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 $\langle n, k \rangle$ to the context, the compiler arranges for an opaque handle $\langle n', k_{\text{com}} \rangle$ (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, $\square_{\text{L}^P}$ 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 $\mathcal{L}$ 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 $\mathcal{L}$. 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 \((n, k)\) is the 4th element of \(L\), its mask is \((4, k_{\text{com}})\). To implement lookup in \(L\) we must compare capabilities too, so we rely on \texttt{eqcap}. To ensure capabilities do not leak to the context, the second field of the pair is a constant capability \(k_{\text{com}}\) 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 \(S\). 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 \(S\). On subsequent ? (incoming) actions, the compiled code will register all new locations available as parameters or reachable from \(S\). Then, on any ! (outgoing) action, the compiled code must scan whatever locations (that the compiled code has created) are now reachable from \(S\) and add them to \(S\). We need the new instructions \texttt{isloc} and \texttt{ispair} in \(LP\) to compute these reachable locations. Of course, this kind of scanning of locations reachable from \(S\) at every call/return between components can be extremely costly.

**Enforcing the Masking of Locations** The functions \texttt{mask} and \texttt{unmask} are added by the compiler to the compiled code. The first function takes a location (which intuitively contains a value \(v\)) and replaces (in \(v\)) any pair \((n, k)\) of a location protected with a component-created capability \(k\) with its index in the masking list \(L\). The second function replaces any pair \((n, k_{\text{com}})\) with the \(n\)th element of the masking list \(L\). These functions should not be directly accessible to the context (else it can \texttt{unmask} any \texttt{mask}'d location and break full abstraction). This is why \(LP\) 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 \texttt{LU}\textsuperscript{U} has a lot of runtime overhead in calculating the heap reachable from \texttt{S} and in masking and unmasking 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 \texttt{S}, \texttt{L} and other supporting data structures to implement the runtime checks described above. In contrast, the code compiled through \texttt{J\cdotKLUP} (which is just robustly safe and not fully abstract) has none of these overheads.

14.3 Proving that \texttt{LU}\textsuperscript{U} is a Fully Abstract Compiler

Using \texttt{LU}\textsuperscript{U} as a concrete example, we now discuss why proving FAC is harder than proving RSC. Consider the hard part of FAC, the forward implication, \( C_1 \simeq_{ctx} C_2 \Rightarrow \llbracket C_1 \rrbracket^S_T \simeq_{ctx} \llbracket C_2 \rrbracket^S_T \). The contrapositive of this statement is \( \llbracket C_1 \rrbracket^S_T \not\simeq_{ctx} \llbracket C_2 \rrbracket^S_T \Rightarrow C_1 \not\simeq_{ctx} C_2 \). By unfolding the definition of \( \not\simeq_{ctx} \) we see that, given a target context \( C \) that distinguishes \( \llbracket C_1 \rrbracket^S_T \) from \( \llbracket C_2 \rrbracket^S_T \), it is necessary to show that there exists a source context \( C \) that distinguishes \( C_1 \) from \( C_2 \). That source context \( C \) must be built (backtranslated) starting from the already given target context \( C \) that differentiates \( \llbracket C_1 \rrbracket^S_T \) from \( \llbracket C_2 \rrbracket^S_T \).

A backtranslation directed by the syntax of the target context \( C \) is hopeless here since the target expressions \texttt{iscap} and \texttt{isloc} cannot be directly backtranslated to valid source expressions. Hence, we resort to another well-known technique \cite{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 if their trace semantics differ in at least one trace. We write \( TR(C) \) to denote the traces of the component \( C \) in this fully abstract semantics. Given this trace semantics, the statement of the forward implication of full abstraction reduces to:

\[
TR(\llbracket C_1 \rrbracket_{LP}^U) \neq TR(\llbracket C_2 \rrbracket_{LP}^U) \Rightarrow C_1 \not\simeq_{ctx} C_2.
\]

The advantage of this formulation over the original one is that now we can construct a distinguishing source context for \( C_1 \) and \( C_2 \) using the trace on which \( TR(\llbracket C_1 \rrbracket_{LP}^U) \) and \( TR(\llbracket C_2 \rrbracket_{LP}^U) \) 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 \( LP \) 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: call read v $H$? · ret $H'$! and call write v $H$? · ret $H'$!. 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 these functions need to be backtranslated to the corresponding source constructs (! and :=, 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 read can be simply ignored since the effects of reads are already captured by later actions in traces. Calls and returns to 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 write.

Example 5 (Backtranslation of traces). Consider the trace below and its backtranslation.

$\begin{array}{ll}
(1) \quad \text{call f 0 1 } \mapsto \text{4?} & \text{main(x) } \mapsto \\
(2) \quad \text{ret 1 } \mapsto \text{⟨1,k_{com}⟩!} & \text{let x = new 4 in L :: ⟨x,1⟩} \\
(3) \quad \text{call write }\langle\langle1,k_{com}\rangle,5\rangle \text{ 1 } \mapsto \text{⟨1,k_{com}⟩?} & \text{call f 0} \\
\text{ret 1 } \mapsto \text{⟨1,k_{com}⟩!} & \text{let x =! L(1) in B :: ⟨x,1⟩} \\
\end{array}$

Note: 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 1) a masked location with index 1 and, later, the context writes 5 to it. The backtranslated code must recognise this pattern and store the location that, in the source, corresponds to the mask 1 in the list $B$ (action 2). In action 3, when it is time to write 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.

93
15 A Fully Abstract Compiler from \(L_U\) to \(L_P\)

We perform the aforementioned changes to languages.

15.1 The Source Language \(L_U\)

In \(L_U\) 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.

\[
\begin{align*}
\text{Components} &\quad C ::= \mathcal{F}; \mathcal{I}; \mathcal{E} \\
\text{Exports} &\quad E ::= f \\
\text{Expressions} &\quad e ::= \cdots | \text{size} \\
\hline
\text{(L\textsuperscript{U}-Size)} &\quad |H| = n \\
\hline
\text{Helpers} &\quad H \triangleright \text{size} \leftrightarrow n
\end{align*}
\]

The semantics is unchanged, it only relies on the new helper functions above.

15.2 The Target Language \(L_P\)

15.2.1 Syntax Changes

\[
\begin{align*}
\text{Components} &\quad C ::= \mathcal{F}; \mathcal{I}; \mathcal{E}; k_{\text{root}}, k_{\text{com}} \\
\text{Exports} &\quad E ::= f
\end{align*}
\]
Expressions $e ::= \cdots | \text{isloc}(e) | \text{ispair}(e) | \text{eqcap}(e, e)$

Trace states $\Theta ::= (C; H; \Pi \triangleright (t) \tau)$

Trace bodies $t ::= s | \text{unk}$

Trace labels $\delta ::= \epsilon | \beta$

Trace actions $\beta ::= \text{call } f v H ? | \text{call } f v H ! | \text{ret } H ! | \text{ret } H ? | \downarrow | \uparrow | \text{write}(v, n)$

Traces $\beta ::= \emptyset | \beta \cdot \beta$

We assume programs are given two capabilities they own: $k_{\text{root}}$ and $k_{\text{com}}$ 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 ($\beta$), to visually distinguish between the two traces and the kind of information carried by them.

We need a write label $\text{write}(v, n)$ that tells that masked location $n$ has been updated to value $v$. 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 additional element $\Pi$, 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 $\text{unk}$ to indicate when the called function was in the context.

Helper functions are as above.

15.2.2 Semantics Changes

In $L^P$ we need functionality to tell if a pair is a location or not and to traverse values in order to extract such locations.

$$(H \triangleright e \leftarrow \langle n, v \rangle \quad n \mapsto \eta \in H \quad \eta = v \text{ or } \eta = \bot) \Rightarrow b = \text{true}$$

$$(H \triangleright \text{isloc}(e) \leftarrow b)$$

$$(H \triangleright e \leftarrow \langle v, v \rangle \Rightarrow b = \text{true}$$

otherwise $b = \text{false}$

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 \( k_{\text{com}} \) 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.

\[
\begin{align*}
\text{(LP-cap-truth)} & \quad H \triangleright e \leftrightarrow k & H \triangleright e' \leftrightarrow k \\
& \quad H \triangleright \text{eqcap}(e, e') \leftrightarrow \text{true} \\
\text{(LP-cap-false)} & \quad H \triangleright e \leftrightarrow k & H \triangleright e' \leftrightarrow k' \quad k \neq k' \\
& \quad H \triangleright \text{eqcap}(e, e') \leftrightarrow \text{false}
\end{align*}
\]

15.2.3 A Fully Abstract Trace Semantics for \( \text{LP} \)

\[
\Theta \xrightarrow{\beta} \Theta' \quad \text{State } \Theta \text{ emits visible action } \beta \text{ becoming } \Theta'.
\]

\[
\Theta \xrightarrow{\bar{\beta}} \Theta' \quad \text{State } \Theta \text{ emits trace } \bar{\beta} \text{ becoming } \Theta'.
\]

**Helper functions**

- **(Reachable)**

\[
\begin{align*}
H \vdash \text{reachable}(n, H') & \quad n \in \text{reach}(n_st, k_st, H) & k_st \in k_{\text{root}} \cup H' \\
& \quad n \mapsto \_ \in H' & \forall n' \mapsto \_ : \_ \in H' & \forall k \in H. k \notin \_ \in H
\end{align*}
\]

- **(Valid heap)**

\[
\begin{align*}
H' = H_{\text{priv}} \cup H_{\text{sha}} & \quad H'' = H_{\text{sha}} \cup H_{\text{new}} \\
\text{dom}(H) = n & \quad \text{dom}(H') = n' \\
H'' \vdash \text{valid}(v', H) & \quad H'' \vdash \text{reachable}(v', H_{\text{priv}} \cup H_{\text{sha}})
\end{align*}
\]

\[
H \vdash \text{validHeap}(H, H', H'', n, n')
\]

- **(LP-Traces-Silent)**

\[
C; \hat{H}; H \triangleright (s_T) \xrightarrow{\bar{\beta}} (C; H'; H \triangleright (s'_T))
\]

\[
(C; H; H \triangleright (s_T) \xrightarrow{\beta} (C; H'; H \triangleright (s'_T))
\]

96
\[
(C; H; \overline{n} ▷ (\text{unk})_F) \xrightarrow{\text{call } f \triangleright H'} (C; H''; \overline{n}'' ▷ (\text{s; return})_F)
\]

\[
(C; H; \overline{n} ▷ (\text{unk})_F) \xrightarrow{\text{call } f \triangleright H'} (C; H''; \overline{n}'' ▷ (\text{skip})_F)
\]

\[
(C; H; \overline{n} ▷ (\text{s})_F) \xrightarrow{\text{call } f \triangleright H'} (C; H; \overline{n}'' ▷ (\text{unk})_F)
\]

\[
\begin{align*}
\forall n. & (C; H; \overline{n} ▷ (s)_F) \xrightarrow{\text{write} w} (C; H; \overline{n'} ▷ (s')_F) \\
& (C; H; \overline{n} ▷ (\text{unk})_F) \xrightarrow{\text{write}(v,n)} (C; H'; \overline{n} ▷ (\text{unk})_F)
\end{align*}
\]
\[\begin{align*}
\alpha \xrightarrow{\text{L.P.-trans}} & \Rightarrow \alpha' \quad \alpha'' \xrightarrow{\text{L.P.-trans}} \Rightarrow \alpha'
\end{align*}\]

\[\begin{align*}
\beta \xrightarrow{\text{L.P.-Traces-Initial}} & \Rightarrow \beta' \\
\text{(L.P.-Traces-Initial)}
\end{align*}\]

\[\begin{align*}
n \in \overline{\mathbb{N}} & \iff n \mapsto v; \eta \in H \quad \text{main} \notin \text{dom}(F) \\
\Theta_0(C) = (C; H; \pi \triangleright (\text{unk}_{\text{main}})
\end{align*}\]

\[\begin{align*}
\text{TR}(C) = \left\{ \beta \mid \Theta_0(C) \xrightarrow{\text{L.P.}} \right\}
\end{align*}\]

15.2.4 Results about the Trace Semantics

The following results hold for \( C_1 = [C_1]_{\text{LP}} \) and \( C_2 = [C_2]_{\text{LP}} \).

**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:

<table>
<thead>
<tr>
<th>Location / creator</th>
<th>( [C_{1}]_{\text{LP}} )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>(1) to ( [C_{1}]_{\text{LP}} )</td>
<td>(2) to ( C )</td>
</tr>
<tr>
<td>Shared</td>
<td>(3) with ( C )</td>
<td>(4) with ( [C_{2}]_{\text{LP}} )</td>
</tr>
</tbody>
</table>

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).

\[\begin{align*}
\text{if } C_1 \simeq_{\text{ctx}} C_2 & \\
\text{then } TR(C_1) = TR(C_2)
\end{align*}\]

**Proof Sketch.** By contraposition:

\[\begin{align*}
\text{if } TR(C_1) \neq TR(C_2) & \\
\text{then } \exists A \cdot A[C_1]_{\text{LP}} \wedge A[C_2]_{\text{LP}} (wlog)
\end{align*}\]

We are thus given \( \beta_1 = \beta \cdot \beta_1 \) and \( \beta_2 = \beta \cdot \beta_2 \) and \( \beta_1 \neq \beta_2 \).

We can construct a context \( A \) that replicates the behaviour of \( \beta \) and then performs the differentiation.

98
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).

\[
\text{if } \text{TR}(C_1) = \text{TR}(C_2) \quad \text{then } C_1 \simeq_{ctx} C_2
\]

Proof Sketch. By contradiction let us assume that

\[ \exists A. A[C_1] \Downarrow \land A[C_2] \Uparrow (wlog) \]

Contexts are deterministic, so they cannot behave differently based on the values of locations that are never shared with \(C_1\) or \(C_2\).

The semantics forbids guessing, so a context will never have access to the locations that \(C_1\) or \(C_2\) do not share.

Thus a context can exhibit a difference in behaviour by relying on something that \(C_1\) modified unlike \(C_2\) 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 `call f v 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 \(C_1\) and \(C_2\) are captured on the labels.

Having reached a contradiction, this case holds.

Lemma 29 (Full abstraction of the trace semantics for compiled components).

\[
\text{TR}(\mathbf{L}_C^{L^U}_P) = \text{TR}(\mathbf{L}_C^{L^U}_P) \iff \mathbf{L}_C^{L^U}_P \simeq_{ctx} \mathbf{L}_C^{L^U}_P
\]

Proof. By Lemmas 27 and 28.
15.3 The Compiler

\[
\begin{align*}
[F; I; E]_{LP} & = \begin{cases} 
[F]_{LP} \mapsto \text{read}(x) \mapsto s_{\text{read}}, \text{write}(x) \mapsto s_{\text{write}}, \\
\quad \text{mask}(x) \mapsto s_{\text{mask}}, \text{unmask}(x) \mapsto s_{\text{unmask}}, \cdots 
\end{cases} \\
\{I\}^U_{LP} \mapsto \\
\{E\}^U_{LP} \mapsto \text{read}, \text{write}; \\
k_{\text{root}}, k_{\text{com}}
\end{align*}
\]

\[
\begin{cases} 
\{f(x) \mapsto s; \text{return}\}^U_{LP} & = f(x) \mapsto s_{\text{add}}(x); \\
\{f\}^U_{LP} & = f \\
\{I\}^U_{LP} & = f
\end{cases}
\]

Expression translation unmodified: \([e]_{LP}^U = [e]_{LP}^U.

Statement translation unmodified except for

\[
\begin{cases} 
\{\text{let } x = \text{new } e \text{ in } s\}^U_{LP} & = \text{let } x_{\text{loc}} = \text{new } [e]_{LP}^U \text{ in } \\
& \quad \text{let } x_{\text{cap}} = \text{hide } x_{\text{loc}} \text{ in } \\
& \quad s_{\text{register}}(x_{\text{loc}}, x_{\text{cap}}); \\
& \quad \text{let } x = \langle x_{\text{loc}}, x_{\text{cap}} \rangle \text{ in } [s]_{LP}^U \\
\{\text{call } f\}^U_{LP} & = \text{let } x = [e]_{LP}^U \text{ in } s_{\text{add}}(x); s_{\text{post}}; \text{call } f \ x; s_{\text{pre}}
\end{cases}
\]

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;

So the compiler is mostly unchanged.
• instead, before passing control to the context, all component-created locations that are shared with the context are masked, i.e., their representation \((n, k)\) is replaced with \((n', k_{\text{com}})\), where \(n'\) 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 \((n', k_{\text{com}})\) that are sub-values of reachable locations with \((n, k)\), which is the \(n'\)th pair in the list of component-allocated locations;

• what is a “component-shared” location? A shared location is a pair \((n, k)\) where (i) \(k\) 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 \((n, _)\) 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 \(n\) and then add it to the list of shared locations at index \(n'\). That location \((n, k)\) would be masked as \((n', k_{\text{com}})\), which grants the context direct access to it. This is where we need to use \(k_{\text{com}}\) 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

\[
\text{let } x = \text{func } v \text{ in s}
\]
to intend: call function \texttt{func} with parameter \texttt{v} and store its returned value in \texttt{x} for use in \texttt{s}.

We indicate how that statement can be expressed in our language with the following desugaring:

\begin{quote}
\texttt{let y = new 0 in let z = \langle v, y \rangle in call func z; let x =!z.2 with 0 in s}
\end{quote}

15.3.2 Support Data Structures

The compiler relies on a number of data structures it keeps starting from location \texttt{0}, which is accessible via \texttt{k\texttt{root}}.

These data structures are:

- a list of capabilities, which we denote with \texttt{K}. These capabilities are those that the compiled component has allocated.

- a list of component-allocated locations, which we denote with \texttt{L}. These are locations \langle \texttt{n}, \texttt{k} \rangle that are created by the compiled component and whose \texttt{k} are elements of \texttt{K}.

- a list of shared locations, which we denote with \texttt{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 \texttt{L} of elements \texttt{e}, we use these helper functions:

- \texttt{indexof(L, e)} returns \texttt{n}, the index of \texttt{e} in \texttt{L}, or \texttt{0} if \texttt{e} is not in \texttt{L};

- \texttt{L(n)} returns the \texttt{n}th element \texttt{e} of \texttt{L} or \texttt{0} if the list length is shorter than \texttt{n};

- \texttt{L :: e} if \texttt{e} is not in \texttt{L}, it adds element \texttt{e} to the list, increasing its length by \texttt{1};

- \texttt{rem(L, e)} removes element \texttt{e} from \texttt{L};

- \texttt{e \in L} returns true or false depending on whether \texttt{e} is in \texttt{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

\[ s_{\text{read}} = \text{let } x_n = x.1.1 \text{ in } \]
\[ \quad \text{let } x_k = x.1.2 \text{ in } \]
\[ \quad \text{let } x_{\text{real}} = \mathbb{L}(x_n) \text{ in } \]
\[ \quad \text{let } x_{\text{dest}} = x.2.1 \text{ in } \]
\[ \quad \text{let } x_{\text{dcap}} = x.2.2 \text{ in } \]
\[ \quad \text{let } x_{\text{val}} = !x_{\text{real}} \text{ with } x_k \text{ in } \]
\[ \quad \text{let } x_{\text{dest}} := x_{\text{val}} \text{ with } x_{\text{dcap}} \]

In order to read a location \( \langle n, k \rangle \), we receive that as the first projection of parameter \( x \). Because we do not explicitly return values, we need the second projection of \( x \) to contain the destination where to target receives the result of the read.

We split the pair in the masking index \( x_n \) and in the capability to access the location \( x_k \). 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

\[ s_{\text{write}} = \text{let } x_n = x.1.1 \text{ in } \]
\[ \quad \text{let } x_k = x.1.2 \text{ in } \]
\[ \quad \text{let } x_{\text{real}} = \mathbb{L}(x_n) \text{ in } \]
\[ \quad x_{\text{real}} := x.2 \text{ with } x_k ; \]

In order to write value \( v \) a location \( \langle n, k \rangle \) we receive a parameter structured as follows: \( x \equiv \langle n, k \rangle, v \). Then we unfold the elements of the parameter and lookup element \( n \) in the list of component-defined locations. We use this looked-up element to write the value \( v \) there.

We do not need to mask \( v \) because it cannot point to locations that are created by the compiled component.

At this stage, \( v \) 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

\[ s_{\text{mask}} = \forall \langle n, k \rangle \in x. \text{ isloc}(\langle n, k \rangle) \]
\[ \quad \text{if } k \in K \]
\[ \quad \text{replace } \langle n, k \rangle \text{ with } (\text{indexof}(\mathbb{L}, n), k_{\text{com}}) \]
We use the abstract construct replace... to indicate the following. We want to keep the value passed as parameter \(x\) unchanged but replace its sub-values 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 \(x \in K\) 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**

\[
s_{\text{unmask}} \triangleq \forall \langle n, k \rangle \in x \\
  \text{if } k == k_{\text{com}} \\
  \text{replace } \langle n, k \rangle \text{ with } L(n)
\]

In the case of unmasking, we receive a value through parameter \(x\) and we know that there may be sub-values of it of the form \(\langle n, k \rangle\) where \(n\) is an index in the component-created shared locations. So we lookup the element from that list and replace it in \(x\).

**15.3.4 Inlined Additional Statements (Preamble, Postamble, etc)**

**Adding**

\[
s_{\text{add}}(x) = \text{if } \text{isloc}(x) \text{ then } \\
  S :: x; \\
  \text{if } x.2 \in K \text{ then } L :: x \text{ else skip}
\]

This common part ensures that the parameter \(x\) 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 \([\text{call } f]_{L^P}\).

**Registration**

\[
s_{\text{register}}(x_{\text{loc}}, x_{\text{cap}}) = K :: x_{\text{cap}};
\]

This statement registers capability \(x_{\text{cap}}\) 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.

\[ s_{\text{pre}} = \forall \langle n, k \rangle \in \text{reach}(\mathcal{S}). \text{isloc}(\langle n, k \rangle) \]
\[ \text{if } \langle n, k \rangle \notin \mathcal{S} \text{ then } \mathcal{S} :: \langle n, k \rangle ; \text{ else skip} \]
\[ \forall \langle n, k \rangle \in \mathcal{S}. \text{isloc}(\langle n, k \rangle) \]
\[ \text{let } x = \text{unmask}(!n \text{ with } k) \text{ in } n := x \text{ with } k \]

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 \texttt{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.

\[ s_{\text{post}}(x) = \forall \langle n, k \rangle \in \text{reach}(\mathcal{S}). \text{isloc}(\langle n, k \rangle) \]
\[ \text{if } \langle n, k \rangle \notin \mathcal{S} \text{ then } \mathcal{S} :: \langle n, k \rangle ; \mathcal{L} :: \langle n, k \rangle ; \text{ else skip} \]
\[ \forall \langle n, k \rangle \in \mathcal{S}. \text{isloc}(\langle n, k \rangle) \]
\[ \text{let } x = \text{mask}(!n \text{ with } k) \text{ in } n := x \text{ with } k \]

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 \texttt{mask} function (line 4).

### 15.4 The Trace-based Backtranslation: \( \llangle \llangle \cdot \rrangle \rrangle^\mathcal{LP}_L^\mathcal{U} \)

Value backtranslation is the same, so \( \llangle \llangle v \rrangle \rrangle^\mathcal{LP}_L^\mathcal{U} = \llangle v \rrangle^\mathcal{LP}_L^\mathcal{U} \).

#### 15.4.1 The Skeleton

The skeleton is almost as before (Section 4.2.2), with the only addition of another list \( \mathcal{B} \) explained below.

The only additions are two functions \texttt{terminate} and \texttt{diverge}, which do what their name suggests:

\[ \text{terminate}(x) \mapsto \text{fail} \]
\[ \text{diverge}(x) \mapsto \text{call diverge 0} \]
15.4.2 The Common Prefix

call f v H? As in Rule (⟨⟨·⟩⟩₁₅₉-\text{call}), we keep a list of the context-allocated locations and we update them. Also, we extend that list.

ret ?H As above.

call f v H! This is analogous to Rule (⟨⟨·⟩⟩₁₅₉-\text{callback-loc}) but with a major complication.

Now this is complex because in the target we don’t receive locations ⟨n, k⟩ from the compiled component, but masked indices ⟨i, k_{com}⟩. (using 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 ℓ ≈ β ⟨n, k⟩ now is no longer true because we have ℓ ≈ β ⟨i, k_{com}⟩ which cannot hold. We need to keep this relation as a runtime argument in the backtranslation and base it solely on the syntactic occurrences of ⟨i, k_{com}⟩. 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 ⟨⣨⣨i⟩⟩₁₅₉,ℓ⟩.

Intuitively, consider heap H from the action. For all of its content n → v : η, we do a structural analysis of v. This happens at the meta-level, in the backtranslation algorithm. v may contain subvalues of the form ⟨i, k_{com}⟩, and accessing this subvalue we know is a matter of ·₁ etc. So we produce an expression e with the same instructions ·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 ⟨i, ℓ⟩ (also given that i = ⟨⣨⣨i⟩⟩₁₅₉).

ret !H As above.

write(v, 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 i so we can produce the correct code: ℓ := ⟨⣨⣨v⟩⟩₁₅₉.

ℓ is looked up as B(⟨⣨⣨i⟩⟩₁₅₉).

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 ℓ₁ (at runtime) and the call stack ̄
We now go over the various cases of trace difference and see that the differentiation code exists. We consider $\alpha_1$ to be the last action in the trace of $[C_1]_T^5$ while $\alpha_2$ is the last one of $[C_2]_T^5$, both made after a common prefix.

$\alpha_1 = \text{call } f \ v \ H!$ and $\alpha_2 = \text{call } g \ v \ H!$. Code if $!\ell_i == i$ then call terminate 0 else skip is placed in the body of $f$ while the code if $!\ell_i == i$ then call diverge 0 else skip is placed in the body of $g$.

$\alpha_1 = \text{call } f \ v \ H!$ and $\alpha_2 = \text{call } f \ w \ H!$. Code

$$\text{if } !\ell == i \text{ then } \text{if } x == \langle\langle v \rangle\rangle_{LP} \text{ then call terminate 0 else call diverge 0 else skip}$$

is placed in the body of $f$.

$\alpha_1 = \text{call } f \ v \ H!$ and $\alpha_2 = \text{call } f \ v \ H'!$. Here few cases can arise, consider $H = H_1, n \mapsto v : \eta, H_2$ and $H' = H_1, n' \mapsto v' : \eta', H_2'$:

$v \neq v'$. We use shortcut $L_{glob}(n)$ to indicate the location bound to name $n$ in the list of shared locations (same as in Section 4.2.3).

Code

$$\text{if } !\ell_i == i \text{ then } \text{let } x = \text{update}(\langle\langle n \rangle\rangle_{LP}, 0) \text{ in call diverge 0 else skip}$$

is placed in the body of $f$.

$n \neq n'$. In this case one of the two addresses must be bigger than the other. Wlog, let’s consider $n = n' + 1$.

So $H_1 = H_1', n' \mapsto v' : \eta'$ and $H_2' = \emptyset$ (otherwise we’d have a binding for $n$ there).

The code in this case must access the location related to $n$, it will get stuck in one case and succeed in the other:

$$\text{if } !\ell_i == i \text{ then let } x = \text{update}(\langle\langle (n) \rangle\rangle_{LP}, 0) \text{ in call diverge 0 else skip}$$

$\eta \neq \eta'$. 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.

$\alpha_1 = \text{ret } H!$ and $\alpha_2 = \text{ret }!$. As above.
\( \alpha_1 = \text{call } f \text{ v } H! \) and \( \alpha_2 = \text{ret} \) ! Code if \( \ell_i = i \) then call terminate 0 else skip is placed at \( f \) while if \( \ell_i = i \) then call diverge 0 else skip is placed at the top of \( f \).

\( \alpha_1 = \text{call } f \text{ v } H! \) and \( \alpha_2 = \downarrow \) Code if \( \ell_i = i \) then call diverge 0 else skip is placed at \( f \).

\( \alpha_1 = \text{call } f \text{ v } H! \) and \( \alpha_2 = \uparrow \) Code if \( \ell_i = i \) then call terminate 0 else skip is placed at the top of \( f \).

\( \alpha_1 = \text{ret } H! \) and \( \alpha_2 = \downarrow \) Code if \( \ell_i = i \) then call diverge 0 else skip is placed at the top of \( f \).

\( \alpha_1 = \text{ret } H! \) and \( \alpha_2 = \uparrow \) Code if \( \ell_i = i \) then call terminate 0 else skip is placed at the top of \( f \).

\( \alpha_1 = \downarrow \) and \( \alpha_2 = \uparrow \) Nothing to do, the compiled component performs the differentiation on its own.
References


