Secure Compilation Lecture Notes*

Marco Patrignani
special thanks to: Akram El-Korashy, Dominique Devriese, Daniel Patterson

Contents

*These lecture notes are the result of the course evolution throughout numerous years. I would like to thank the students of the following courses that used preliminary versions of these lecture notes for their feeback and endurance: (sc101 @ Cispa&UdS'18), (cs350 @ Stanford'18), (scs19 @ Cispa&UdS'19), (cs350 @ Stanford'19)

1 Pure Whole Languages

1.1 Source Language S^{w}

1.1.1 Syntax

$$
\begin{aligned}
\mathsf{P} &::= \mathsf{e} \\
\mathsf{e} &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{e} \odot \mathsf{e} \mid \mathsf{n} \mid \mathsf{x} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e} \mid \mathsf{fail} \\
\mathsf{v} &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{n} \\
\mathsf{f} &::= \mathsf{n} \mid \mathsf{fail} \\
\mathsf{E} &::= [\cdot] \mid \mathsf{e} \oplus \mathsf{E} \mid \mathsf{E} \oplus \mathsf{n} \mid \mathsf{let}\ \mathsf{x} = \mathsf{E}\ \mathsf{in}\ \mathsf{e} \mid \mathsf{if}\ \mathsf{E}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e} \mid \mathsf{e} \odot \mathsf{E} \mid \mathsf{E} \odot \mathsf{n} \\
\gamma &::= \varnothing \mid \gamma; [\mathsf{v} \,/\, \mathsf{x}]
\end{aligned}
$$

The program state is just an expression e.

We assume some standard properties of substitutions γ, such as capture avoidance, distributivity over terms, and weakening.

In the following, we assume: $\oplus ::= +, -, \cdots$, $\odot ::= <, >, ==$.

1.1.2 Dynamic Semantics

Judgements: $\mathsf{e} \hookrightarrow^0 \mathsf{e}'$ $\quad$ $\mathsf{e} \hookrightarrow \mathsf{e}'$

$$
\frac{}{\mathsf{let}\ \mathsf{x} = \mathsf{n}\ \mathsf{in}\ \mathsf{e} \hookrightarrow^0 \mathsf{e}[\mathsf{n} \,/\, \mathsf{x}]}\ (\text{Eval-letin}) \qquad \frac{n \oplus n' = n''}{\mathsf{n} \oplus \mathsf{n}' \hookrightarrow^0 \mathsf{n}''}\ (\text{Eval-op}) \qquad \frac{n \odot n' = v}{\mathsf{n} \odot \mathsf{n}' \hookrightarrow^0 \mathsf{v}}\ (\text{Eval-bop})
$$

$$
\frac{}{\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e}' \hookrightarrow^0 \mathsf{e}}\ (\text{Eval-if-t}) \qquad \frac{}{\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ \mathsf{e}\ \mathsf{else}\ \mathsf{e}' \hookrightarrow^0 \mathsf{e}'}\ (\text{Eval-if-f})
$$

$$
\frac{\mathsf{e} \hookrightarrow^0 \mathsf{e}'}{\mathsf{E}[\mathsf{e}] \hookrightarrow \mathsf{E}[\mathsf{e}']}\ (\text{Eval-ctx}) \qquad \frac{}{\mathsf{E}[\mathsf{fail}] \hookrightarrow \mathsf{fail}}\ (\text{Eval-fail})
$$

We can use evaluation contexts (which are not program contexts) to determine where the reductions happen.

1.1.3 Static Semantics

$$
\begin{aligned}
\tau &::= \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{Nat} \to \mathsf{Nat} \\
\Gamma &::= \varnothing \mid \Gamma; \mathsf{x} : \tau
\end{aligned}
$$

$\Gamma \vdash \mathsf{e} : \tau$ $\quad$ Well-typed expression e of type τ

$\vdash \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e} : \mathsf{Nat} \to \mathsf{Nat}$ $\quad$ Well-typed program e of type Nat to Nat

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ (\text{T-true}) \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ (\text{T-false}) \qquad \frac{}{\Gamma \vdash \mathsf{n} : \mathsf{Nat}}\ (\text{T-n}) \qquad \frac{\mathsf{x} : \tau \in \Gamma}{\Gamma \vdash \mathsf{x} : \tau}\ (\text{T-x})$$

$$\frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{Nat} \quad \Gamma \vdash \mathsf{e}_2 : \mathsf{Nat}}{\Gamma \vdash \mathsf{e}_1 \odot \mathsf{e}_2 : \mathsf{Bool}}\ (\text{T-bop}) \qquad \frac{\Gamma \vdash \mathsf{e}_1 : \mathsf{Nat} \quad \Gamma \vdash \mathsf{e}_2 : \mathsf{Nat}}{\Gamma \vdash \mathsf{e}_1 \oplus \mathsf{e}_2 : \mathsf{Nat}}\ (\text{T-op}) \qquad \frac{\Gamma \vdash \mathsf{e}_1 : \tau_1 \quad \Gamma; \mathsf{x} : \tau_1 \vdash \mathsf{e}_2 : \tau}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathsf{in}\ \mathsf{e}_2 : \tau}\ (\text{T-let})$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{Bool} \quad \Gamma \vdash \mathsf{e}_1 : \tau \quad \Gamma \vdash \mathsf{e}_2 : \tau}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{e}_1\ \mathsf{else}\ \mathsf{e}_2 : \tau}\ (\text{T-if}) \qquad \frac{}{\Gamma \vdash \mathsf{fail} : \tau}\ (\text{T-fail})$$

$$\frac{\mathsf{x} : \mathsf{Nat} \vdash \mathsf{e} : \mathsf{Nat} \quad \mathsf{fail} \notin \mathsf{e}}{\vdash \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e} : \mathsf{Nat} \to \mathsf{Nat}}\ (\text{T-prog})$$

1.2 Target Language T^{w}

1.2.1 Syntax

$$\begin{aligned}
\mathbf{P} &::= \mathbf{e} \\
\mathbf{v} &::= \mathbf{n} \mid \mathbf{true} \mid \mathbf{false} \qquad \mathbf{b} ::= \mathbf{true} \mid \mathbf{false} \\
\mathbf{e} &::= \mathbf{n} \mid \mathbf{x} \mid \mathbf{e} \oplus \mathbf{e} \mid \mathbf{let}\ \mathbf{x} = \mathbf{e}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{fail} \\
&\quad \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{e} \odot \mathbf{e} \mid \mathbf{e}\ \mathbf{has}\ \mathbf{T} \\
\mathbf{T} &::= \mathbf{Nat} \mid \mathbf{Bool} \\
\mathbf{f} &::= \mathbf{n} \mid \mathbf{fail} \\
\mathbf{E} &::= [\cdot] \mid \mathbf{e} \oplus \mathbf{E} \mid \mathbf{E} \oplus \mathbf{v} \mid \mathbf{let}\ \mathbf{x} = \mathbf{E}\ \mathbf{in}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{E}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \mid \mathbf{e} \odot \mathbf{E} \mid \mathbf{E} \odot \mathbf{v} \\
\gamma &::= \varnothing \mid \gamma; [\mathbf{v} \,/\, \mathbf{x}]
\end{aligned}$$

1.2.2 Dynamic Semantics

Judgements: $\mathbf{e} \hookrightarrow^{0} \mathbf{e}' \qquad \mathbf{e} \hookrightarrow \mathbf{e}'$

$$\frac{}{\mathbf{let}\ \mathbf{x} = \mathbf{v}\ \mathbf{in}\ \mathbf{e} \hookrightarrow^{0} \mathbf{e}[\mathbf{v} \,/\, \mathbf{x}]}\ (\text{Eval-letin}) \qquad \frac{n \oplus n' = n''}{\mathbf{n} \oplus \mathbf{n}' \hookrightarrow^{0} \mathbf{n}''}\ (\text{Eval-op}) \qquad \frac{n \odot n' = v}{\mathbf{n} \odot \mathbf{n}' \hookrightarrow^{0} \mathbf{v}}\ (\text{Eval-bop})$$

$$\frac{}{\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e}' \hookrightarrow^{0} \mathbf{e}}\ (\text{Eval-if-t}) \qquad \frac{}{\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e}' \hookrightarrow^{0} \mathbf{e}'}\ (\text{Eval-if-f})$$

$$\frac{\mathbf{e} \hookrightarrow^{0} \mathbf{e}'}{\mathbf{E}[\mathbf{e}] \hookrightarrow \mathbf{E}[\mathbf{e}']}\ (\text{Eval-ctx}) \qquad \frac{}{\mathbf{E}[\mathbf{fail}] \hookrightarrow \mathbf{fail}}\ (\text{Eval-fail}) \qquad \frac{}{\mathbf{n}\ \mathbf{has}\ \mathbf{N} \hookrightarrow^{0} \mathbf{true}}\ (\text{Eval-has-nN})$$

$$\frac{}{\mathbf{n}\ \mathbf{has}\ \mathbf{B} \hookrightarrow^{0} \mathbf{false}}\ (\text{Eval-has0nB}) \qquad \frac{\mathbf{v} \equiv \mathbf{true}\ \text{or}\ \mathbf{false}}{\mathbf{v}\ \mathbf{has}\ \mathbf{N} \hookrightarrow^{0} \mathbf{false}}\ (\text{Eval-hasvN}) \qquad \frac{}{\mathbf{v}\ \mathbf{has}\ \mathbf{B} \hookrightarrow^{0} \mathbf{true}}\ (\text{Eval-has-vB})$$

$$\frac{\mathbf{v} = \mathbf{b} \text{ or } \mathbf{v}' = \mathbf{b}}{\mathbf{v} \oplus \mathbf{v}' \hookrightarrow^0 \mathbf{fail}}\ (\text{Eval-op-fail}) \qquad \frac{\mathbf{v} = \mathbf{b} \text{ or } \mathbf{v}' = \mathbf{b}}{\mathbf{v} \odot \mathbf{v}' \hookrightarrow^0 \mathbf{fail}}\ (\text{Eval-bop-fail})$$

$$\frac{}{\mathbf{if\ n\ then\ e\ else\ e}' \hookrightarrow^0 \mathbf{fail}}\ (\text{Eval-if-fail})$$

1.3 $[\![\cdot]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$: Compiler from $\mathsf{S^w}$ to $\mathbf{T^w}$

$$\begin{aligned}
[\![\mathsf{true}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{true} \\
[\![\mathsf{false}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{false} \\
[\![\mathsf{n}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{n} \\
[\![\mathsf{x}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{x} \\
[\![\mathsf{e} \oplus \mathsf{e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \oplus [\![\mathsf{e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \\
[\![\mathsf{e} \odot \mathsf{e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \odot [\![\mathsf{e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \\
[\![\mathsf{let\ x = e\ in\ e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{let\ x =} [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \mathbf{\ in\ } [\![\mathsf{e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \\
[\![\mathsf{if\ b\ then\ e\ else\ e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{if\ } [\![\mathsf{b}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \mathbf{\ then\ } [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \mathbf{\ else\ } [\![\mathsf{e}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \\
[\![\mathsf{fail}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \mathbf{fail} \\
[\![[\mathsf{v}\ /\ \mathsf{x}]]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} &= \left[[\![\mathsf{v}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \Big/ [\![\mathsf{x}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \right]
\end{aligned}$$

1.4 Compiler Correctness for $[\![\cdot]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$

Lemma 1.1 (Forward simulation).

$$\text{if } \mathsf{e}\gamma \hookrightarrow^* \mathsf{v} \text{ then } [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} [\![\gamma]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \hookrightarrow^* [\![\mathsf{v}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$$

Find the proof template in Appendix A.1.

Lemma 1.2 (Expression correctness).

$$\text{if } [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} [\![\gamma]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \hookrightarrow^* [\![\mathsf{f}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \text{ then } \mathsf{e}\gamma \hookrightarrow^* \mathsf{f}$$

Proof. By contradiction we assume: $\mathsf{e}\gamma \hookrightarrow \mathsf{v}' \neq \mathsf{v}$ (the case for b is analogous).

By Lemma 1.1 (Forward simulation) we get that $[\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} [\![\gamma]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \hookrightarrow^* [\![\mathsf{v}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$.

By determinism of the compiler we have $[\![\mathsf{v}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \neq [\![\mathsf{v}']\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$.

So we have that the same term $[\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$ reduces to two different terms, which contradicts the determinism of the semantics. □

Theorem 1.3 (Compiler correctness for $[\![\cdot]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$ (Whole Programs)).

$$\textit{if } \varnothing \vdash \mathsf{P} : \tau \textit{ and } [\![\mathsf{P}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \hookrightarrow^* [\![\mathsf{f}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \textit{ then } \mathsf{P} \hookrightarrow^* \mathsf{f}$$

Proof. By Lemma 1.2 (Expression correctness). □

2 Partial Languages

2.1 Source Language Additions: S_τ

$$\mathsf{P} ::= \mathsf{fn(x)} \mapsto \mathsf{e}$$
$$\mathsf{C} ::= \mathsf{let\ y = call\ fn\ e\ in\ e}$$

$\vdash \mathsf{let\ y = call\ fn\ e_1\ in\ e_2 : Bool}$ Well-typed context of type Bool

$$\frac{\begin{array}{c}\text{(T-ctx)} \\ \varnothing \vdash \mathsf{e_1 : Nat} \\ \mathsf{y : Nat} \vdash \mathsf{e_2 : Bool}\end{array}}{\vdash \mathsf{let\ y = call\ fn\ e_1\ in\ e_2 : Bool}}$$

2.2 Target Language Additions: $\mathbf{T_u}$

$$\mathbf{P} ::= \mathbf{fn(x)} \mapsto \mathbf{e}$$
$$\mathbf{C} ::= \mathbf{let\ y = call\ fn\ e\ in\ e}$$

2.3 Common Definitions

Definition 2.1 (Contextual equivalence).

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall C.\ C\,[P_1] \hookrightarrow^* f \text{ and } C\,[P_2] \hookrightarrow^* f$$

Definition 2.2 (Plugging). Given that $C = \mathit{let}\ y = \mathit{call}\ \mathit{fn}\ e_{arg}\ \mathit{in}\ e_{cont}$ and $P = fn(x) \mapsto e_{fun}$.

$$C\,[P] \stackrel{\text{def}}{=} \mathit{let}\ y = \mathit{let}\ x = e_{arg}\ \mathit{in}\ e_{fun}\ \mathit{in}\ e_{cont}$$

Example 2.3 (Equivalent and inequivalent programs).

$\mathsf{x + 2}$	$\mathsf{x + 1 + 1}$
$\mathsf{if\ x > 0\ then\ 0\ else\ 1}$	x
$\mathsf{let\ z = x\ in\ z + z}$	$\mathsf{x + x}$

⊡

2.4 $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$: Compiler from S_τ to $\mathbf{T_u}$

$$
\begin{aligned}
[\![\mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} &= \mathbf{fn}(\mathbf{x}) \mapsto [\![\mathsf{e}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \\
[\![\mathsf{e}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} &= [\![\mathsf{e}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \\
[\![\mathsf{let\ y = call\ fn\ e\ in\ e'}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} &= \mathbf{let\ y = call\ fn}\ [\![\mathsf{e}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}\ \mathbf{in}\ [\![\mathsf{e'}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}
\end{aligned}
$$

2.5 Compiler Correctness for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$

Theorem 2.4 (Compiler correctness for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$ (Partial Programs)).

if $\varnothing \vdash \mathsf{P} : \tau$ *and* $\vdash \mathsf{C} : \mathsf{Bool}$ *and* $\mathbf{C} = [\![\mathsf{C}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$ *and* $\mathbf{C}\left[[\![\mathsf{P}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}\right] \hookrightarrow^* [\![\mathsf{f}]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$
then $\mathsf{C}[\mathsf{P}] \hookrightarrow^* \mathsf{f}$

Proof. Analogous to the proof of Theorem 1.3 (Compiler correctness for $[\![\cdot]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$ (Whole Programs)).

3 Compiler Security: Fully Abstract Compilation

Theorem 3.1 (Full abstraction of $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$).

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$$

Example 3.2 (Violations of full abtraction).

$$
\begin{aligned}
\mathbf{C_b} &= \mathbf{let\ y = call\ fn\ true\ in\ 2} \\
\mathsf{P}_1 &= \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{let\ z = x\ in\ 1} \\
\mathsf{P}_2 &= \mathsf{fn}(\mathsf{x}) \mapsto 1
\end{aligned}
$$

In this case, we have $\forall \mathsf{C}, \exists \mathsf{f}$:

- $\mathsf{C}[\mathsf{P}_1] \hookrightarrow^* \mathsf{f}$
- $\mathsf{C}[\mathsf{P}_2] \hookrightarrow^* \mathsf{f}$
- $\mathbf{C_b}\left[[\![\mathsf{P}_1]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}\right] \hookrightarrow^* \mathbf{fail}$
- $\mathbf{C_b}\left[[\![\mathsf{P}_2]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}\right] \hookrightarrow^* \mathbf{2}$

So, $\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$ but $[\![\mathsf{P}_1]\!]^{\mathsf{S^w}}_{\mathbf{T^w}} \not\simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S^w}}_{\mathbf{T^w}}$. ⊡

3.1 Compiler Changes

$$[\![\mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} = \mathbf{fn(x) \mapsto \text{if } x \text{ has } N \text{ then } [\![e]\!]^{S_\tau}_{T_u} \text{ else fail}}$$

3.2 Fully Abstract Compilation via Context-based Back-translation

3.2.1 Reflection

Lemma 3.3 (Equivalence reflection for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$).

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Leftarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$$

Proof. We state this in contrapositive form:

$$\mathsf{P}_1 \not\simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \not\simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$$

and expand the definitions of $\simeq_{ctx}$:

$$\begin{aligned}&\exists \mathsf{C}.\ \mathsf{C}[\mathsf{P}_1] \hookrightarrow^* \mathsf{f} \ \textit{and}\ \mathsf{C}[\mathsf{P}_2] \hookrightarrow^* \mathsf{f}' \ \textit{and}\ \mathsf{f} \neq \mathsf{f}'\\ &\Rightarrow \exists \mathbf{C}.\ \mathbf{C}\left[[\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}\right] \hookrightarrow^* \mathbf{f} \ \textit{and}\ \mathbf{C}\left[[\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}\right] \hookrightarrow^* \mathbf{f}' \ \textit{and}\ \mathbf{f} \neq \mathbf{f}'\end{aligned}$$

Picking $\mathbf{C}$ is simple, assuming $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$ can be applied to context (as is generally the case, like here), $\mathbf{C}{=}[\![\mathsf{C}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$. At this point, a clever usage of Lemma 1.1 (Forward simulation) gives this result. □

3.2.2 Preservation

Lemma 3.4 (Equivalence preservation for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$).

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$$

Proof attempt (this proof is not completed).. We state this in contrapositive form:

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \not\simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \Rightarrow \mathsf{P}_1 \not\simeq_{ctx} \mathsf{P}_2$$

and expand the definitions of $\simeq_{ctx}$:

$$\begin{aligned}\forall \mathsf{P}_1, \mathsf{P}_2.\ \ &\exists \mathbf{C}.\ \mathbf{C}\left[[\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}\right] \hookrightarrow^* \mathbf{f} \ \textit{and}\ \mathbf{C}\left[[\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}\right] \hookrightarrow^* \mathbf{f}' \ \textit{and}\ \mathbf{f} \neq \mathbf{f}'\\ &\Rightarrow \exists \mathsf{C}.\ \mathsf{C}[\mathsf{P}_1] \hookrightarrow^* \mathsf{f} \ \textit{and}\ \mathsf{C}[\mathsf{P}_2] \hookrightarrow^* \mathsf{f}' \ \textit{and}\ \mathsf{f} \neq \mathsf{f}'\end{aligned}$$

We can try to build C starting from $\mathbf{C}$ as we cannot rely on any correctness result.

3.3 $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$: Context-based Backtranslation

3.3.1 The Universal Type

We need a universal type, something to backtranslate target expression to in order for them to be valid.

Example 3.5 (Backtranslation type). We cannot backtranslate $\mathbf{true}$ to true because when backtranslating $\mathbf{3 + true}$ we would get $\mathsf{3 + true}$ that is not a valid source expression according to the grammar of a. Also, we need a mechanism that scales for all operations quantified over by $\oplus$, e.g., $\mathbf{3 * 2}$ etc. ⊡

Anything that the target is backtranslated to, must be of this universal type.

This universal type is *natural numbers*.

3.3.2 Helper Functions

Then we need to convert to and from normal types and the universal type in order to ensure proper communication occurs. In fact, if we backtranslate $\mathbf{call\ f\ true}$ to $\mathsf{call\ f\ 0}$, the former will fail (by the typecheck inserted by the compiler) and the second will not.

Inject takes something of a type and injects it into the universal type, extract takes from the universal type and extracts to a type.

$$\mathsf{inject_{Nat}(e) = e + 2}$$
$$\mathsf{inject_{Bool}(e) = if\ e\ then\ 1\ else\ 0}$$
$$\mathsf{extract_{Nat}(e) = let\ x = e\ in\ if\ x \geq 2\ then\ x - 2\ else\ fail}$$
$$\mathsf{extract_{Bool}(e) = let\ x = e\ in\ if\ x \geq 2\ then\ fail\ else\ if\ x - 1 \geq 1\ then\ false\ else\ true}$$

3.3.3 Properties of the Helpers

Lemma 3.6 (The Helpers are well-typed). The following holds:

- If $\Gamma \vdash \mathsf{e : Nat}$ then $\Gamma \vdash \mathsf{inject_{Nat}(e) : Nat}$
- If $\Gamma \vdash \mathsf{e : Bool}$ then $\Gamma \vdash \mathsf{inject_{Bool}(e) : Nat}$
- If $\Gamma \vdash \mathsf{e : Nat}$ then $\Gamma \vdash \mathsf{extract_{Nat}(e) : Nat}$
- If $\Gamma \vdash \mathsf{e : Nat}$ then $\Gamma \vdash \mathsf{extract_{Bool}(e) : Bool}$

Proof. Simple case analysis.

3.3.4 The Backtranslation

The backtranslation is based on the program context structure.

$$\langle\!\langle \mathbf{let\ y = call\ fn\ e'\ in\ e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} = \mathsf{let\ y{=}inject_{Nat}(call\ fn\ (extract_{Nat}}\langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\mathsf{))\ in\ } \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$$

$$
\begin{aligned}
\langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{n} + 2 \\
\langle\!\langle \mathbf{x} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{x} \\
\langle\!\langle \mathbf{true} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= 0 \\
\langle\!\langle \mathbf{false} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= 1 \\
\langle\!\langle \mathbf{e} \oplus \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{let\ x1{=}extract_{Nat}} \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in} \\
&\quad \mathsf{let\ x2{=}extract_{Nat}} \langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in} \\
&\quad \mathsf{inject_{Nat} x1} \oplus \mathsf{x2} \\
\langle\!\langle \mathbf{e} \odot \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{let\ x1{=}extract_{Nat}} \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in} \\
&\quad \mathsf{let\ x2{=}extract_{Nat}} \langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in} \\
&\quad \mathsf{inject_{Bool} x1} \oplus \mathsf{x2} \\
\langle\!\langle \mathbf{let\ x = e\ in\ e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{let\ x} = \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in\ } \langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \\
\langle\!\langle \mathbf{if\ e\ then\ e'\ else\ e''} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{let\ x} = \mathsf{extract_{Bool}} \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in\ if\ x\ then\ } \langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ else\ } \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \\
\langle\!\langle \mathbf{e\ has\ T} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \begin{cases} \mathsf{let\ x} = \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in\ if\ x} \geq 2 \mathsf{\ then\ 1\ else\ 0} \\ \quad \text{if } \mathbf{T} = \mathbf{Nat} \\ \mathsf{let\ x} = \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \mathsf{\ in\ if\ x} \geq 2 \mathsf{\ then\ 0\ else\ 1} \\ \quad \text{if } \mathbf{T} = \mathbf{Bool} \end{cases} \\
\langle\!\langle \mathbf{fail} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \mathsf{fail} \\
\langle\!\langle [\mathbf{v} \,/\, \mathbf{x}] \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} &= \left[\langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \,/\, \langle\!\langle \mathbf{x} \rangle\!\rangle^{\mathbf{T_u}}_{S_\tau} \right]
\end{aligned}
$$

The case for the $\mathbf{e\ has\ T}$ from should be $\mathsf{inject_{Bool}(if\ x} \geq 2 \mathsf{\ then\ true\ else\ false)}$ (in the case for Bool, swap true and false for the Nat case) but we shorten it to the definition above because we already know how the "if" expression and the subsequent inject will execute.

Remark 3.7 (Letin). The backtranslation of $\mathbf{let\ x = e\ in\ e'}$ may seem confusing, as it does not insert $\mathsf{inject_{Nat}}$ for its subexpressions.

We argue why it is right using this example, for which we indicate the reductions:

- $\mathbf{e_1 = let\ x = 2\ in\ x + 1}$ and $\mathbf{t_1 \hookrightarrow^* 3}$;

Currently, what we get is:

- $\mathsf{e_1 = let\ x = 4\ in\ let\ x1 = extract_{Nat} x\ in\ let\ x2 = extract_{Nat} 3\ in\ inject_{Nat} x1 + x2}$

If we unfold the reductions, we see that

$$
\begin{array}{rl}
e_1 & \mathsf{let\ x = 4\ in\ let\ x1 = extract_{Nat} x\ in\ let\ x2 = extract_{Nat} 3\ in\ inject_{Nat} x1 + x2} \\
& \hookrightarrow \mathsf{let\ x1 = extract_{Nat} 4\ in\ let\ x2 = extract_{Nat} 3\ in\ inject_{Nat} x1 + x2} \\
& \hookrightarrow^* \mathsf{let\ x2 = extract_{Nat} 3\ in\ inject_{Nat} 2 + x2} \\
& \hookrightarrow^* \mathsf{inject_{Nat} 2 + 1} \\
& \hookrightarrow^* \mathsf{5}
\end{array}
$$

and these reductions proceed as expected.

However if we insert an additional $\mathsf{extract_{Nat}}$ for the value bound to the x, these reductions will not go right, as we get an additional $+2$. We could eliminate it by adding an $\mathsf{inject_{Nat}}$ when variables are backtranslated, but this is hard to do correctly as we do not know if a variable will be used as a Nat or as a Bool in the target, as in this other valid expression:

- $\mathbf{e_2 = let\ x = true\ in\ if\ x\ then\ 3\ else\ 0}$

This gets backtranslated to

- $\mathsf{e_2 = let\ x = 0\ in\ let\ z = extract_{Bool} x\ in\ if\ z == 1\ then\ 5\ else\ 2}$

These expressions reduce correctly, but we would not know how to carry the information that $\mathbf{x}$ is technically a Boolean (the $\mathbf{true}$ expression may be a lot more complex than that and provide no help).

However, we know that *when a variable is going to be used*, e.g., inside a $\oplus$ expression, the $\mathsf{extract.}$ will be there.

3.3.5 Properties of the Backtranslation

In order to use the context backtranslation, we need to prove that it is correct:

Lemma 3.8 (Backtranslation correctness).

$$
\begin{array}{c}
\text{if } \mathbf{e}\gamma \hookrightarrow^* \mathbf{f} \\
\text{then } \langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \langle\!\langle \gamma \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \hookrightarrow^* \langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}
\end{array}
$$

Find the proof template in Appendix A.2.

3.3.6 Fully Abstract Compilation Reflection

We resume our proof for the $\Rightarrow$ direction of fully abstract compilation.

Proof of Lemma 3.4. What we have is:

$$
\begin{array}{l}
\exists \mathbf{C}.\ \mathbf{C}\left[\llbracket \mathsf{P_1} \rrbracket^{\mathsf{S}_\tau}_{\mathbf{T_u}}\right] \hookrightarrow^* \mathbf{f} \text{ and } \mathbf{C}\left[\llbracket \mathsf{P_2} \rrbracket^{\mathsf{S}_\tau}_{\mathbf{T_u}}\right] \hookrightarrow^* \mathbf{f}' \text{ and } \mathbf{f} \neq \mathbf{f}' \\
\Rightarrow \exists \mathsf{C}.\ \mathsf{C}[\mathsf{P_1}] \hookrightarrow^* \mathsf{f} \text{ and } \mathsf{C}[\mathsf{P_2}] \hookrightarrow^* \mathsf{f}' \text{ and } \mathsf{f} \neq \mathsf{f}'
\end{array}
$$

We can instantiate C with $\langle\!\langle \mathbf{C} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$.

So we can assume:

1. $\mathbf{C}\left[\left[\!\left[\mathsf{P}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\right] \hookrightarrow^* \mathbf{f}$
2. $\mathbf{C}\left[\left[\!\left[\mathsf{P}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\right] \hookrightarrow^* \mathbf{f}'$

and prove this

- $\langle\!\langle \mathbf{C} \rangle\!\rangle^{\mathbf{T}_u}_{\mathsf{S}_\tau}[\mathsf{P}_1] \hookrightarrow^* \mathsf{f}$
- $\langle\!\langle \mathbf{C} \rangle\!\rangle^{\mathbf{T}_u}_{\mathsf{S}_\tau}[\mathsf{P}_2] \hookrightarrow^* \mathsf{f}'$

If we unfold the definition of $C\,[P]$ in the hypotheses, assuming that $\mathsf{P}_1 = \mathsf{fn(x)} \mapsto \mathsf{e}_1$ and $\mathsf{P}_2 = \mathsf{fn(x)} \mapsto \mathsf{e}_2$ we obtain

1. $\mathbf{let\ y = let\ x = e'\ in\ if\ x\ has\ NAT\ then}\ \left[\!\left[\mathsf{e}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ \mathbf{else\ fail\ in\ e''} \hookrightarrow^* \mathbf{f}$
2. $\mathbf{let\ y = let\ x = e'\ in\ if\ x\ has\ NAT\ then}\ \left[\!\left[\mathsf{e}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ \mathbf{else\ fail\ in\ e''} \hookrightarrow^* \mathbf{f}'$

We can unfold the reductions to see that:

$$\left[\!\left[\mathsf{P}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u} \left|\begin{array}{l} \mathbf{let\ y = let\ x = e'\ in\ if\ x\ has\ NAT\ then}\ \left[\!\left[\mathsf{e}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ \mathbf{else\ fail\ in\ e''} \\ \hookrightarrow^* \mathbf{let\ y = let\ x = v'\ in\ if\ x\ has\ NAT\ then}\ \left[\!\left[\mathsf{e}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ \mathbf{else\ fail\ in\ e''} \\ \hookrightarrow \mathbf{let\ y = if\ v'\ has\ NAT\ then}\ (\left[\!\left[\mathsf{e}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}[\mathbf{v'}\ /\ \mathbf{x}])\ \mathbf{else\ fail\ in\ e''} \end{array}\right.$$

We know that $\mathbf{e'}$ must reduce to a value and not to **fail** because otherwise both target executions would reduce to **fail**, contradicting the hypothesis $\mathbf{f} \neq \mathbf{f}'$.

Also, by determinism and given that the program context is the same in both cases, we know that until here, the reductions are the same for the second program too, so:

$$\left[\!\left[\mathsf{P}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u} \left|\begin{array}{l} \mathbf{let\ y = let\ x = e'\ in\ if\ x\ has\ NAT\ then}\ \left[\!\left[\mathsf{e}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ \mathbf{else\ fail\ in\ e''} \\ \hookrightarrow^* \mathbf{let\ y = if\ v'\ has\ NAT\ then}\ (\left[\!\left[\mathsf{e}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}[\mathbf{v'}\ /\ \mathbf{x}])\ \mathbf{else\ fail\ in\ e''} \end{array}\right.$$

Now we can perform a case analysis on $\mathbf{v'}$ and rule out the case for booleans, otherwise both target executions would reduce to **fail**, contradicting the hypothesis $\mathbf{f} \neq \mathbf{f}'$.

So $\mathbf{v'}$ really is a natural number $\mathbf{n'}$:

$$\left[\!\left[\mathsf{P}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u} \left|\begin{array}{l} \mathbf{let\ y = if\ n'\ has\ NAT\ then}\ (\left[\!\left[\mathsf{e}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}[\mathbf{n'}\ /\ \mathbf{x}])\ \mathbf{else\ fail\ in\ e''} \\ \hookrightarrow \mathbf{let\ y = (}\left[\!\left[\mathsf{e}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}[\mathbf{n'}\ /\ \mathbf{x}])\ \mathbf{in\ e''} \\ \hookrightarrow^* \mathbf{let\ y = (}\left[\!\left[\mathsf{n}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\mathbf{)\ in\ e''} \\ \hookrightarrow \mathbf{e''}\left[\left[\!\left[\mathsf{n}_1\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ /\ \mathbf{y}\right] \\ \hookrightarrow^* \mathbf{f} \end{array}\right.$$

The execution of P_2, instead, must differ, so, determinism of the semantics let us conclude that these reductions happen, for a $\mathsf{n}_2 \neq \mathsf{n}_1$:

$$\left[\!\left[\mathsf{P}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u} \left|\begin{array}{l} \mathbf{let\ y = if\ n'\ has\ NAT\ then}\ (\left[\!\left[\mathsf{e}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}[\mathbf{n'}\ /\ \mathbf{x}])\ \mathbf{else\ fail\ in\ e''} \\ \hookrightarrow^* \mathbf{e''}\left[\left[\!\left[\mathsf{n}_2\right]\!\right]^{\mathsf{S}_\tau}_{\mathbf{T}_u}\ /\ \mathbf{y}\right] \\ \hookrightarrow^* \mathbf{f'} \end{array}\right.$$

Let us take a look at the source reductions. By Lemma 3.8 (Backtranslation correctness), we know the following:

$$\mathsf{P}_1 \left| \begin{array}{l} \quad \mathsf{let\ y = inject_{Nat}(let\ x = extract_{Nat}}\langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\ \mathsf{in\ e_1)\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \\ \hookrightarrow^* \mathsf{let\ y = inject_{Nat}(let\ x = extract_{Nat}}\langle\!\langle \mathbf{v'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\ \mathsf{in\ e_1)\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \end{array} \right.$$

By inspecting the target reductions we know, $\mathsf{extract_{Nat}}\langle\!\langle \mathbf{v'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$ cannot fail, as $\mathbf{v'}$ is a natural number. Additionally, we know that $\mathsf{v'}{=}\langle\!\langle \mathbf{n'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} - 2$. By Lemma 1.2 (Expression correctness) we know:

$$\mathsf{P}_1 \left| \begin{array}{l} \quad \mathsf{let\ y = inject_{Nat}e_1[v'\ /\ x]\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\left[\langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\ /\ \mathsf{z}\right] \\ \hookrightarrow^* \mathsf{let\ y = inject_{Nat}n_1\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \end{array} \right.$$

So by definition of $\mathsf{inject_{Nat}}$ we have that $\mathsf{n'_1 = n_1 + 2}$:

$$\mathsf{P}_1 \left| \begin{array}{l} \quad \mathsf{let\ y = inject_{Nat}n_1\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \\ \hookrightarrow \mathsf{let\ y = n'_1\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \\ \hookrightarrow^* \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}[\mathsf{n'_1\ /\ y}] \end{array} \right.$$

So we need to show the reductions for P_2, again by Lemma 3.8 (Backtranslation correctness):

$$\mathsf{P}_2 \left| \begin{array}{l} \quad \mathsf{let\ y = inject_{Nat}(let\ x = extract_{Nat}}\langle\!\langle \mathbf{e'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\ \mathsf{in\ e_2)\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \\ \hookrightarrow^* \mathsf{let\ y = inject_{Nat}a_2[v'\ /\ x]\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \end{array} \right.$$

And by how the reductions proceded in the target and Lemma 1.2 (Expression correctness), we know that:

$$\mathsf{P}_2 \left| \begin{array}{l} \hookrightarrow \mathsf{let\ y = inject_{Nat}n_2\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \end{array} \right.$$

We can define $\mathsf{n'_2 = n_2 + 2}$ and we get:

$$\mathsf{P}_2 \left| \begin{array}{l} \hookrightarrow \mathsf{let\ y = n'_2\ in}\ \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \\ \hookrightarrow \langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}[\mathsf{n'_2\ /\ y}] \end{array} \right.$$

At this stage in the proof we make a simplification for the sake of clarity. We lift this and show how to conclude this proof properly in Section 3.3.7.

For simplicity, we can assume that $\mathbf{e''}$ is $\mathbf{if\ y == [\![n1]\!]^{S_\tau}_{T_u}\ then\ 0\ else\ 1}$ and thus $\mathbf{f = 0}$ and $\mathbf{f' = 1}$.

Given that we know $\mathbf{e''}$, let us work out its backtranslation (modulo some optimisation and elimination of bits that we know how will reduce):

P_1 & P_2

$$
\begin{aligned}
&\text{let } z = \mathsf{extract}_{\mathsf{Bool}}\langle\!\langle \mathbf{y} == \mathbf{n_1} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \text{ in if } z == 1 \text{ then } \langle\!\langle \mathbf{0} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \text{ else } \langle\!\langle \mathbf{1} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \\
=\ &\text{let } z = \mathsf{extract}_{\mathsf{Bool}} \text{ let } x1{=}\mathsf{extract}_{\mathsf{Nat}}\langle\!\langle \mathbf{y} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \text{ in } \quad \text{in if } z == 1 \text{ then } 2 \text{ else } 3 \\
&\qquad\qquad \text{let } x2{=}\mathsf{extract}_{\mathsf{Nat}}\langle\!\langle \mathbf{n_1} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \text{ in} \\
&\qquad\qquad \mathsf{inject}_{\mathsf{Bool}} x1 == x2 \\
=\ &\text{let } z = \mathsf{extract}_{\mathsf{Bool}} \text{ let } x1{=}\mathsf{extract}_{\mathsf{Nat}} y \text{ in } \qquad \text{in if } z == 1 \text{ then } 2 \text{ else } 3 \\
&\qquad\qquad \text{let } x2{=}\mathsf{extract}_{\mathsf{Nat}} n_1 + 2 \text{ in} \\
&\qquad\qquad \mathsf{inject}_{\mathsf{Bool}} x1 == x2 \\
=\ &\text{let } z = \mathsf{extract}_{\mathsf{Bool}} \text{ let } x1{=}y - 2 \text{ in } \qquad \text{in if } z == 1 \text{ then } 2 \text{ else } 3 \\
&\qquad\qquad \text{let } x2{=}n_1 \text{ in} \\
&\qquad\qquad \mathsf{inject}_{\mathsf{Bool}} x1 == x2 \\
=\ &\text{if } y - 2 == n_1 \text{ then } 2 \text{ else } 3
\end{aligned}
$$

So we have

P_1

$$
\begin{aligned}
&\text{if } y - 2 == n_1 \text{ then } 2 \text{ else } 3[n_1' \,/\, y] \\
=\ &\text{if } n_1' - 2 == n_1 \text{ then } 2 \text{ else } 3 \\
=\ &\text{if } n_1 + 2 - 2 == n_1 \text{ then } 2 \text{ else } 3 \\
\hookrightarrow\ &2
\end{aligned}
$$

Now we know $\langle\!\langle \mathbf{e''} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$ and we know that $n_2 \neq n_1$, so we have that

P_2

$$
\begin{aligned}
&= \text{if } y - 2 == n_1 \text{ then } 2 \text{ else } 3[n_2' \,/\, y] \\
&= \text{if } n_2 + 2 - 2 == n_1 \text{ then } 2 \text{ else } 3 \\
&= \text{if } n_2 == n_1 \text{ then } 2 \text{ else } 3 \\
&\hookrightarrow 3
\end{aligned}
$$

So $f{=}2$ and $f'{=}3$ and this proof holds. □

3.3.7 Proper Differentiation

Note that the assumption on what $\mathbf{e''}$ looks like is a simplifying assumption just for the sake of explanation. To conclude the proof properly we need to apply again Lemma 3.8 and we need the following trivial lemma too:

Lemma 3.9 (Differentiation).

$$\text{if } \mathbf{f} \neq \mathbf{f'} \text{ then } \langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \neq \langle\!\langle \mathbf{f'} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$$

Proof. Trivial case analyisis.

Proper completion of Lemma 3.4. We have:

- $\mathbf{e''}\left[[\![n_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \,/\, \mathbf{y}\right] \hookrightarrow^* \mathbf{f'}$
- $\mathbf{e''}\left[[\![n_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \,/\, \mathbf{y}\right] \hookrightarrow^* \mathbf{f'}$

and we need to reason about

- $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}[\mathsf{n}'_1 \,/\, \mathsf{y}]$
- $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}[\mathsf{n}'_2 \,/\, \mathsf{y}]$

Since $\mathsf{n}'_1 = \mathsf{n}_1 + 2$, $\mathsf{n}'_2 = \mathsf{n}_2 + 2$, we can rewrite the last two items as

- $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\left[\langle\!\langle \mathbf{n_1} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \,/\, \langle\!\langle \mathbf{y} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\right]$
- $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\left[\langle\!\langle \mathbf{n_2} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \,/\, \langle\!\langle \mathbf{y} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\right]$

By how backtranslation works on substitutions, we can apply Lemma 3.8 (Backtranslation correctness) twice to conclude that:

- $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\left[\langle\!\langle \mathbf{n_1} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \,/\, \langle\!\langle \mathbf{y} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\right] \hookrightarrow^* \langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$
- $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\left[\langle\!\langle \mathbf{n_2} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \,/\, \langle\!\langle \mathbf{y} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}\right] \hookrightarrow^* \langle\!\langle \mathbf{f}' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$

Finally, we apply Lemma 3.9 (Differentiation) to conclude $\langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \neq \langle\!\langle \mathbf{f}' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$ □

3.3.8 Completing the Full Abstraction Proof

Proof of Theorem 3.1. By Lemma 3.4 (Equivalence preservation for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$) and Lemma 3.3 (Equivalence reflection for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$). □

3.4 Fully Abstract Compilation via Trace-based Backtranslation

3.4.1 Target Changes: $\mathbf{T_r}$

$$\mathbf{e} ::= \cdots \mid \mathbf{refl\ e}$$

$$\frac{\mathbf{n} = \|\mathbf{e}\|}{\mathbf{refl\ e} \hookrightarrow \mathbf{n}}\ (\text{Eval-refl})$$

Function $\|\cdot\|$ returns a hash of its argument so each term has its numerical representation.

Notice that this addition is only in the target. The compiler does not generate this expression, so only the context can use it. Moreover, by structure of the context, it can only use that expression on its own expression, not on the programs.

3.4.2 Target Trace Semantics

$$\mathbf{t} ::= \mathbf{call\ n?} \cdot \mathbf{ret\ n!}$$

Definition 3.10 (Traces of a program).

$$\mathbf{TR(P)} = \{\mathbf{call\ n?} \cdot \mathbf{ret\ n'!} \mid \mathbf{P} = \mathbf{fn(x)} \mapsto \mathbf{e} \textit{ and } \mathbf{let\ x = n\ in\ e} \hookrightarrow^* \mathbf{n'}\}$$

3.4.3 Trace Equivalence

Definition 3.11 (Trace equivalence).

$$\mathbf{P_1} \overset{\mathrm{T}}{=} \mathbf{P_2} \overset{\text{def}}{=} \mathbf{TR}(\mathbf{P_1}) = \mathbf{TR}(\mathbf{P_2})$$

Lemma 3.12 (Soundness of traces).

$$\mathbf{P_1} \overset{\mathrm{T}}{=} \mathbf{P_2} \Rightarrow \mathbf{P_1} \simeq_{ctx} \mathbf{P_2}$$

Find the proof in Appendix A.3.

3.4.4 Compiler from S_τ to $\mathbf{T_r}$

$$[\![\mathsf{P}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_r}} = [\![\mathsf{P}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$$

3.5 $\langle\!\langle\!\langle \cdot \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau}$: Trace-based Backtranslation

At this point we are given two different traces which, by definition, agree on the **call**.. parameters and we must build both a source program context C that leads to the differentiation.

$$\mathbf{t_1} = \mathbf{call}\ \mathbf{n}?\cdot\mathbf{ret}\ \mathbf{n^1}! \qquad \mathbf{t_2} = \mathbf{call}\ \mathbf{n}?\cdot\mathbf{ret}\ \mathbf{n^2}!$$

$$\langle\!\langle\!\langle \mathbf{n} \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau} = \mathsf{n}$$

$$\langle\!\langle\!\langle \mathbf{t_1}, \mathbf{t_2} \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau} = \mathsf{let\ y = call\ fn}\ \langle\!\langle\!\langle \mathbf{n} \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau}\ \mathsf{in\ if\ y ==}\ \langle\!\langle\!\langle \mathbf{n^1} \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau}\ \mathsf{then\ 1\ else\ 2}$$

3.5.1 Properties of the Backtranslation of Traces

Lemma 3.13 (Correctness of the backtranslation of traces).

$$\begin{aligned} &\text{if } \mathbf{call}\ \mathbf{n}?\cdot\mathbf{ret}\ \mathbf{n'}! \in \mathbf{TR}([\![\mathsf{P}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}) \\ &\quad \mathsf{P} = \mathsf{fn(x)} \mapsto \mathsf{e} \\ &\text{then } \mathsf{let\ y = let\ x =}\ \langle\!\langle\!\langle \mathbf{n} \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau}\ \mathsf{in\ e\ in\ e'} \hookrightarrow^* \mathsf{e'}\left[\langle\!\langle\!\langle \mathbf{n'} \rangle\!\rangle\!\rangle^{\mathbf{T_r}}_{\mathsf{S}_\tau} \Big/ \mathsf{y}\right] \end{aligned}$$

Proof. This follows by unfolding the definitions of the trace semantics and by Lemma 1.2 (Expression correctness). □

3.5.2 Fully Abstract Compilation

Theorem 3.14 (Full abstraction of $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_r}}$).

$$\forall \mathsf{P_1}, \mathsf{P_2}.\ \ \mathsf{P_1} \simeq_{ctx} \mathsf{P_2} \iff [\![\mathsf{P_1}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_r}} \simeq_{ctx} [\![\mathsf{P_2}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_r}}$$

Proof. Since the compiler has not changed, the reflection holds by Lemma 3.3 (Equivalence reflection for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$). The preservation holds by Lemma 3.15 (Equivalence preservation for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T_r}}$) below. □

3.5.3 Fully Abstract Compilation Preservation

Lemma 3.15 (Equivalence preservation for $[\![\cdot]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}}$).

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}}$$

Proof. We can apply Lemma 3.12 (Soundness of traces) and we get:

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} \overset{\mathbb{T}}{=} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}}$$

in contrapositive form:

$$\forall \mathsf{P}_1, \mathsf{P}_2.\ \ [\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} \overset{\mathbb{T}}{\neq} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} \Rightarrow \mathsf{P}_1 \not\simeq_{ctx} \mathsf{P}_2$$

expanding the definitions:

$$\begin{aligned}\forall \mathsf{P}_1, \mathsf{P}_2.\ \ \exists \mathbf{t} \in \mathbf{TR}([\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}})\ \textit{and}\ \mathbf{t} \notin \mathbf{TR}([\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}}) \\ \Rightarrow \exists \mathsf{C}.\ \mathsf{C}[\mathsf{P}_1] \hookrightarrow^* \mathsf{n}'\ \textit{and}\ \mathsf{C}[\mathsf{P}_2] \hookrightarrow^* \mathsf{n}''\end{aligned}$$

We pick another trace $\mathbf{t}'$ from $\mathbf{TR}([\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}})$ such that the first part is the same as in $\mathbf{t}$ for the backtranslation. Knowing that $[\![\mathsf{n}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} = \mathbf{n^1}$ and $\mathbf{n^1} \neq \mathbf{n^2}$, the traces are

$$\mathbf{t} = \mathbf{call\ n?} \cdot \mathbf{ret\ n^1!} \qquad\qquad \mathbf{t}' = \mathbf{call\ n?} \cdot \mathbf{ret\ n^2!}$$

We can now use the backtranslation of traces with $\mathbf{t}$ and $\mathbf{t}'$ to instantiate C.

The reductions proceed as follows. By Lemma 3.13 (Correctness of the backtranslation of traces) we know:

$$\begin{aligned}[\![\mathsf{P}_1]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} \Big| \quad & \mathsf{let\ y = call\ fn}\ \langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{in\ if\ y ==}\ \langle\!\langle\langle \mathbf{n^1} \rangle\!\rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{then\ 1\ else\ 2} \\ \hookrightarrow^* & \mathsf{if\ y ==}\ \langle\!\langle\langle \mathbf{n^1} \rangle\!\rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{then\ 1\ else\ 2}\left[\langle\!\langle \mathbf{n_1} \rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau} \Big/ \mathsf{y}\right]) \\ \equiv & \mathsf{if\ n_1 ==}\ \langle\!\langle\langle \mathbf{n^1} \rangle\!\rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{then\ 1\ else\ 2}\end{aligned}$$

and

$$\begin{aligned}[\![\mathsf{P}_2]\!]^{\mathsf{S}_\tau}_{\mathbf{T}_{\mathbf{r}}} \Big| \quad & \mathsf{(let\ y = call\ fn}\ \langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{in\ if\ y ==}\ \langle\!\langle\langle \mathbf{n^2} \rangle\!\rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{then\ 1\ else\ 2} \\ \hookrightarrow^* & \mathsf{if\ n_2 = 1 ==}\ \langle\!\langle\langle \mathbf{n^2} \rangle\!\rangle\!\rangle^{\mathbf{T}_{\mathbf{r}}}_{\mathsf{S}_\tau}\ \mathsf{then\ 1\ else\ 2}\end{aligned}$$

The different reductions now are straightforward, so $\mathsf{f} = 1$ and $\mathsf{f}' = 2$ so this case holds. □

4 Stateful Languages

4.1 Source Heap: S_ℓ

4.1.1 Syntax

$$\mathsf{P} ::= \mathsf{H} \triangleright \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e}$$

$$
\begin{aligned}
e &::= \cdots \mid \mathsf{let\ x = new\ e\ in\ e} \mid \mathsf{let\ x = read\ e\ in\ e} \mid \mathsf{let\ x = write\ e\ at\ e\ in\ e} \\
v &::= v \mid \ell \\
E &::= \cdots \mid \mathsf{let\ x = new\ E\ in\ e} \mid \mathsf{let\ x = read\ E\ in\ e} \mid \mathsf{let\ x = write\ E\ at\ e\ in\ e} \\
&\quad \mid \mathsf{let\ x = write\ v\ at\ E\ in\ e} \\
H &::= \varnothing \mid H; \ell \mapsto n \\
\Omega &::= H \triangleright e
\end{aligned}
$$

Programs have a global heap.

$H(\ell)$ returns the number n in H to which ℓ maps to.

$H \cup \ell \mapsto n$ updates H with a (possibly new) binding of the form $\ell \mapsto n$. If ℓ is in the domain of H, the binding for ℓ is updated to point to n.

4.1.2 Dynamic Semantics

Judgement: $\Omega \hookrightarrow \Omega'$

$$
\frac{H \triangleright e \hookrightarrow^0 H \triangleright e'}{H \triangleright e \hookrightarrow H \triangleright e'}\ (\text{Eval-ctx}) \qquad \frac{\ell \notin H}{H \triangleright \mathsf{let\ x = new\ n\ in\ e} \hookrightarrow^0 H \cup \ell \mapsto n \triangleright e[x \,/\, \ell]}\ (\text{Eval-new})
$$

$$
\frac{}{H \triangleright \mathsf{let\ x = read}\ \ell\ \mathsf{in\ e} \hookrightarrow^0 H \triangleright e[H(\ell) \,/\, x]}\ (\text{Eval-read})
$$

$$
\frac{}{H \triangleright \mathsf{let\ x = write\ n\ at}\ \ell\ \mathsf{in\ e''} \hookrightarrow^0 H \cup \ell \mapsto n \triangleright e''[n \,/\, x]}\ (\text{Eval-write})
$$

The new context rule replaces the old one, and any old rule needs to be updated to carry around the heap.

4.1.3 Static Semantics

Programs are now passed locations as input from the context. The related typing rules are trivially changed accordingly.

$$
\begin{aligned}
\tau &::= \cdots \mid \mathsf{Ref\ Nat} \\
\Sigma &::= \varnothing \mid \Sigma; \ell : \mathsf{Ref\ Nat}
\end{aligned}
$$

$\vdash H \triangleright \mathsf{fn}(x) \mapsto e : \mathsf{Ref\ Nat} \to \mathsf{Nat}$ Well-typed program e of type $\mathsf{Ref\ Nat}$ to Nat

$$
\frac{\Gamma, \Sigma \vdash e : \mathsf{Nat} \qquad \Gamma; x : \mathsf{Ref\ Nat}, \Sigma \vdash e' : \tau}{\Gamma, \Sigma \vdash \mathsf{let\ x = new\ e\ in\ e'} : \tau}\ (\text{T-alloc})
$$

$$
\frac{\Gamma, \Sigma \vdash e : \mathsf{Ref\ Nat} \qquad \Gamma; x : \mathsf{Nat}, \Sigma \vdash e' : \tau}{\Gamma, \Sigma \vdash \mathsf{let\ x = read\ e\ in\ e'} : \tau}\ (\text{T-read})
$$

$$\frac{\Gamma, \Sigma \vdash \mathsf{e} : \mathsf{Nat} \quad \Gamma, \Sigma \vdash \mathsf{e}' : \mathsf{Ref\ Nat} \quad \Gamma; \mathsf{x} : \mathsf{Nat}, \Sigma \vdash \mathsf{e}'' : \tau}{\Gamma, \Sigma \vdash \mathsf{let\ x} = \mathsf{write\ e\ at\ e}' \mathsf{\ in\ e}'' : \tau}\ (\text{T-write})$$

$$\frac{\ell \in \mathtt{dom}\,(\Sigma)}{\Gamma, \Sigma \vdash \ell : \mathsf{Ref\ Nat}}\ (\text{T-loc}) \qquad \frac{\mathsf{x} : \mathsf{Nat}, \Sigma \vdash \mathsf{e} : \mathsf{Nat} \quad \mathsf{fail} \notin \mathsf{e} \quad \Sigma = \mathtt{dom}\,(\mathsf{H}) : \mathsf{Ref\ Nat}}{\vdash \mathsf{H} \triangleright \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e} : \mathsf{Ref\ Nat} \to \mathsf{Nat}}\ (\text{T-prog})$$

4.2 Target Heap: $\mathbf{T_n}$

4.2.1 Syntax

$$\begin{aligned}
\mathbf{P} &::= \mathbf{H} \triangleright \mathbf{fn(x)} \mapsto \mathbf{e} \\
\mathbf{e} &::= \cdots \mid \mathbf{let\ x = new\ e\ in\ e} \mid \mathbf{let\ x = read\ e\ in\ e} \mid \mathbf{let\ x = write\ e\ at\ e\ in\ e} \\
\mathbf{E} &::= \cdots \mid \mathbf{let\ x = new\ E\ in\ e} \mid \mathbf{let\ x = read\ E\ in\ e} \\
&\quad \mid \mathbf{let\ x = write\ E\ at\ e\ in\ e} \mid \mathbf{let\ x = write\ v\ at\ E\ in\ e} \\
\mathbf{H} &::= \varnothing \mid \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} \\
\boldsymbol{\Omega} &::= \mathbf{H} \triangleright \mathbf{e}
\end{aligned}$$

4.2.2 Dynamic Semantics

Judgement: $\boldsymbol{\Omega} \hookrightarrow \boldsymbol{\Omega}'$

$$\frac{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow^{0} \mathbf{H} \triangleright \mathbf{e}'}{\mathbf{H} \triangleright \mathbf{e} \hookrightarrow \mathbf{H} \triangleright \mathbf{e}'}\ (\text{Eval-ctx})$$

$$\frac{\text{the cardinality of } \mathbf{H} = n}{\mathbf{H} \triangleright \mathbf{let\ x = new\ v\ in\ e} \hookrightarrow \mathbf{H} \cup \mathbf{n} + \mathbf{1} \mapsto \mathbf{v} \triangleright \mathbf{e}[\mathbf{x} \,/\, \mathbf{n} + \mathbf{1}]}\ (\text{Eval-new})$$

$$\frac{\mathbf{n} \mapsto \mathbf{v} \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{let\ x = read\ n\ in\ e} \hookrightarrow \mathbf{H} \triangleright \mathbf{e}[\mathbf{v} \,/\, \mathbf{x}]}\ (\text{Eval-read})$$

$$\frac{\mathbf{n} \notin \mathtt{dom}\,(\mathbf{H})}{\mathbf{H} \triangleright \mathbf{let\ x = read\ n\ in\ e} \hookrightarrow \mathbf{H} \triangleright \mathbf{e}[\mathbf{false} \,/\, \mathbf{x}]}\ (\text{Eval-read-no})$$

$$\frac{\mathbf{n} \mapsto \mathbf{v} \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{let\ x = write\ v'\ at\ n\ in\ e}'' \hookrightarrow \mathbf{H} \cup \mathbf{n} \mapsto \mathbf{v}' \triangleright \mathbf{e}''[\mathbf{n} \,/\, \mathbf{x}]}\ (\text{Eval-write})$$

Reading always succeeds, but if the location is not allocated, **false** is returned.

4.3 Common Definitions

Assume both languages also contain pairs $\langle e, e \rangle$ and projections $e.1$, $e.2$ as standardly done.

Also, we will use a shorthand $e_1; e_2$ for the expression $let\ \_ = e_1\ in\ e_2$.

Definition 4.1 (Plugging).

$$\begin{aligned} &if\ C = let\ y = call\ fn\ e\ in\ e_c \\ &and\ P = H \triangleright fn(x) \mapsto e_{fun} \\ &then\ C\,[P] \stackrel{\text{def}}{=} H \triangleright let\ y = let\ x = e\ in\ e_{fun}\ in\ e_c \end{aligned}$$

4.3.1 Behaviours

We define a behaviour as a sequence of call/returns from the context to the program. The behaviours of a program is the set of all behaviours it can generate.

Note that while this is analogous to traces, this is for whole programs. Also, while traces capture a sort of context/program interaction, behaviours capture a whole program/environment interaction, e.g., they would capture I/O if our language had any.

$$b ::= call\ n\ H? \cdot ret\ n\ H!$$

While we write this in black, there are really two behaviours, as the heaps are different between S_ℓ and $\mathbf{T_n}$. This will be relevant later.

$$\begin{aligned} C[P] \rightsquigarrow\ & call\ n_1\ H_1? \cdot ret\ n_1'\ H_1'! \\ & \text{if}\ \ C = let\ y_1 = call\ fn\ e_1\ in\ e_c \\ & \quad P = H_0 \triangleright fn(x) \mapsto eb \\ & \quad\quad H_0 \triangleright e_1 \hookrightarrow^* H_1 \triangleright n_1 \\ & \quad\quad H_1 \triangleright let\ x = n_1\ in\ eb \hookrightarrow^* H_1' \triangleright n_1' \\ & \quad\quad H_1' \triangleright let\ y_1 = n_1'\ in\ e_2 \hookrightarrow^* H \triangleright n \end{aligned}$$

$$\mathtt{Behav}\,(C[P]) \stackrel{\text{def}}{=} \{b \mid C[P] \rightsquigarrow b\}$$

4.4 $[\![\cdot]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}$: Compiler from S_ℓ to $\mathbf{T_n}$

The compiler maps a source location to a target number.

$$\begin{aligned} &\cdots \\ [\![\mathsf{H} \triangleright \mathsf{fn(x)} \mapsto \mathsf{e}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= [\![\mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} \triangleright \mathbf{fn(z)} \mapsto \mathbf{let\ x = read\ z\ in} \\ &\qquad\qquad \mathbf{if\ x\ has\ Nat\ then}\ [\![\mathsf{e}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}\ \mathbf{else\ fail} \\ [\![\mathsf{let\ x= new\ e\ in\ e'}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= \mathbf{let\ x_l = new}\ [\![\mathsf{e}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}\ \mathbf{in}\ [\![\mathsf{e'}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} \\ [\![\mathsf{let\ x= read\ e\ in\ e'}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= \mathbf{let}\ [\![\mathsf{x}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} = \mathbf{read}\ [\![\mathsf{e}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}\ \mathbf{in}\ [\![\mathsf{e'}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} \\ [\![\mathsf{let\ x= write\ e\ at\ e'\ in\ e''}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= \mathbf{let}\ [\![\mathsf{x}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} = \mathbf{write}\ [\![\mathsf{e}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}\ \mathbf{at}\ [\![\mathsf{e'}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}\ \mathbf{in}\ [\![\mathsf{e''}, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} \end{aligned}$$

$$
\begin{aligned}
[\![\ell, \mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= \mathbf{n} \\
&\quad \textit{where } \texttt{indexof}\,(\ell, \mathsf{H}) = n \\
[\![\varnothing]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= \varnothing \\
[\![\mathsf{H}; \ell \mapsto \mathsf{n}']\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} &= [\![\mathsf{H}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}; \mathbf{n+1} \mapsto [\![\mathsf{n}', \varnothing]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} \\
&\quad \textit{where } \texttt{card}\,(\mathbf{H}) = n
\end{aligned}
$$

We keep the heap around to know what number to compile a location to. Also, we compile the program global heap giving each location its position in the heap list.

4.4.1 Compiler Shortcomings

Even by typechecking the argument, the context can still guess a location and write a boolean to it.

$$
\begin{aligned}
\mathsf{P}_1 &= \ell \mapsto 2 \triangleright \mathsf{fn(x)} \mapsto \mathsf{let\ y = read\ } \ell \mathsf{\ in\ y + 1; 3} \\
\mathsf{P}_2 &= \ell \mapsto 2 \triangleright \mathsf{fn(x)} \mapsto 3 \\
\mathbf{C_b} &= \mathbf{let\ y = call\ fn\ let\ z = write\ true\ at\ 1\ in\ z\ in\ true}
\end{aligned}
$$

Obviously $\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$.

The reductions of $[\![\mathsf{P}_1]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}$ will get stuck when doing $[\![\mathsf{y}+1]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}$ since $[\![\mathsf{y}]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}$ will contain the **true** value written there by $\mathbf{C_b}$. On the other hand, the reductions of $[\![\mathsf{P}_2]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}$ will not get stuck, so $[\![\mathsf{P}_1]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}} \not\simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}_\ell}_{\mathbf{T_n}}$.

5 Compiler Security: Robustly Safe Compilation

5.1 Target Memory Protection: Capabilities $\mathbf{T_k}$

In this case we do not extend the language $\mathbf{T_n}$, but the language $\mathbf{T_u}$ into the new language $\mathbf{T_k}$.

5.1.1 Syntax

$$
\begin{aligned}
\mathbf{v} &::= \cdots \mid \mathbf{k} \\
\mathbf{e} &::= \cdots \mid \mathbf{let\ x= new\ e\ in\ e} \mid \mathbf{let\ x= read\ e\ with\ e\ in\ e} \\
&\quad \mid \mathbf{let\ x= write\ e\ at\ e\ with\ e\ in\ e} \mid \mathbf{let\ x= hide\ e\ in\ e} \\
\mathbf{C} &::= \mathbf{let\ y = call\ fn\ e\ in\ e} \text{ such that } \mathbf{let\ x= hide\ e\ in\ e} \notin \mathbf{e} \\
\mathbf{E} &::= \cdots \mid \mathbf{let\ x= new\ E\ in\ e} \mid \mathbf{let\ x= read\ E\ with\ e\ in\ e} \\
&\quad \mid \mathbf{let\ x= write\ E\ at\ e\ with\ e\ in\ e} \mid \mathbf{let\ x= write\ v\ at\ E\ with\ e\ in\ e} \\
&\quad \mid \mathbf{let\ x= hide\ E\ in\ e} \\
\mathbf{H} &::= \varnothing \mid \mathbf{H; n} \mapsto \mathbf{n} : \eta \mid \mathbf{H; k}
\end{aligned}
$$

$$\eta ::= \bot \mid \mathbf{k}$$
$$\mathbf{\Omega} ::= \mathbf{H} \triangleright \mathbf{e}$$

5.1.2 Dynamic Semantics

Judgement: $\mathbf{\Omega} \hookrightarrow \mathbf{\Omega}'$

$$\frac{\text{(Eval-new)}\quad \text{if the cardinality of } \mathbf{H} = n}{\mathbf{H} \triangleright \mathbf{let\ x{=}\ new\ n'\ in\ e} \hookrightarrow \mathbf{H} \cup \mathbf{n+1} \mapsto \mathbf{n'} \triangleright \mathbf{e}[\mathbf{x} \,/\, \mathbf{n+1}]}$$

$$\frac{\text{(Eval-read)}\quad \mathbf{n} \mapsto \mathbf{v} : \mathbf{k} \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{let\ x{=}\ read\ n\ with\ k\ in\ e} \hookrightarrow \mathbf{H} \triangleright \mathbf{e}[\mathbf{v} \,/\, \mathbf{x}]}$$

$$\frac{\text{(Eval-read-nocap)}\quad \mathbf{n} \mapsto \mathbf{v} : \bot \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{let\ x{=}\ read\ n\ with\ v\ in\ e} \hookrightarrow \mathbf{H} \triangleright \mathbf{e}[\mathbf{v} \,/\, \mathbf{x}]}$$

$$\frac{\text{(Eval-read-no)}\quad \mathbf{n} \notin \mathtt{dom}\,(\mathbf{H})}{\mathbf{H} \triangleright \mathbf{let\ x{=}\ read\ n\ with\ v\ in\ e} \hookrightarrow \mathbf{H} \triangleright \mathbf{e}[\mathbf{false} \,/\, \mathbf{x}]}$$

$$\frac{\text{(Eval-write)}\quad \mathbf{n} \mapsto \mathbf{v} : \mathbf{k} \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{let\ x{=}\ write\ n'\ at\ n\ with\ k\ in\ e''} \hookrightarrow \mathbf{H} \cup \mathbf{n} \mapsto \mathbf{n'} \triangleright \mathbf{e''}[\mathbf{n} \,/\, \mathbf{x}]}$$

$$\frac{\text{(Eval-write-nocap)}\quad \mathbf{n} \mapsto \_ : \bot \in \mathbf{H}}{\mathbf{H} \triangleright \mathbf{let\ x{=}\ write\ n'\ at\ n\ with\ v\ in\ e''} \hookrightarrow \mathbf{H} \cup \mathbf{n} \mapsto \mathbf{n'} \triangleright \mathbf{e''}[\mathbf{n} \,/\, \mathbf{x}]}$$

5.1.3 $[\![\cdot]\!]^{\mathsf{S}_\ell}_{\mathbf{T_k}}$: Compiler from S_ℓ to $\mathbf{T_k}$

The compiler translates source locations ℓ into pairs $\mathbf{n}, \mathbf{k}$ of the natural number $\mathbf{n}$ which is the target location and a capability $\mathbf{k}$ used to hide location $\mathbf{n}$.

Thus, since programs expect locations, it expects a pair as input. It will treat the first element of the pair as a location and try to read there with the second element. Note that if the location is not allocated, or if it contains a boolean, $\mathbf{x}$ will not pass the dynamic typecheck. Also, note that in the case of programs, the compiler remaps x as $\mathbf{z}$ in order to perform the typecheck on inputs.

$$\beta ::= \varnothing \mid \beta; \ell \mapsto \langle \mathbf{n}, \mathbf{k} \rangle$$

$$\begin{aligned} \mathtt{getbeta}\,(\varnothing) &= \varnothing \\ \mathtt{getbeta}\,(\mathsf{H}; \ell \mapsto \mathsf{n}') &= \mathtt{getbeta}\,(\mathsf{H}); \ell \mapsto \langle \mathbf{n+1}, \mathbf{k} \rangle \\ &\quad \textit{where } \mathtt{card}\,(\mathbf{H}) = n \textit{ and } \mathbf{k} \notin \mathtt{getbeta}\,(\mathsf{H}).\textit{capabilities} \end{aligned}$$

Function $\texttt{getbeta}\,(\mathsf{H})$ inputs a heap and returns a partial bijection β where all locations in the heap are bound to increasing numbers and fresh capabilities.

$$[\![\mathsf{H} \triangleright \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = [\![\mathsf{H}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} \triangleright \mathbf{fn(z)} \mapsto \mathbf{let\ x = read\ z.1\ with\ z.2\ in}\ \mathbf{if\ x\ has\ Nat\ then}\ [\![\mathsf{e}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}\ \mathbf{else\ fail}$$
$$\textit{and}\ \beta = \texttt{getbeta}\,(\mathsf{H})$$

$$\ldots$$

$$[\![\mathsf{let\ x = new\ e\ in\ e'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \mathbf{let\ x_l = new}\ [\![\mathsf{e}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}\ \mathbf{in}\ \mathbf{let\ x_c = hide\ x_l\ in}\ \mathbf{let}\ [\![\mathsf{x}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \langle \mathbf{x_l}, \mathbf{x_c} \rangle\ \mathbf{in}\ [\![\mathsf{e'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}$$
$$[\![\mathsf{let\ x = read\ e\ in\ e'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \mathbf{let\ x_p} = [\![\mathsf{e}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}\ \mathbf{in}\ \mathbf{let}\ [\![\mathsf{x}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \mathbf{read\ x_p.1\ with\ x_p.2\ in}\ [\![\mathsf{e'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}$$
$$[\![\mathsf{let\ x = write\ e\ at\ e'\ in\ e''}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \mathbf{let\ x_p} = [\![\mathsf{e'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}\ \mathbf{in}\ \mathbf{let}\ [\![\mathsf{x}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \mathbf{write}\ [\![\mathsf{e}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}\ \mathbf{at\ x_p.1\ with\ x_p.2\ in}\ [\![\mathsf{e'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}$$
$$[\![\ell, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \beta\,(\ell)$$
$$[\![\varnothing; \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = \varnothing$$
$$[\![\mathsf{H}; \ell \mapsto \mathsf{n'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} = [\![\mathsf{H}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k}; \mathbf{n} \mapsto [\![\mathsf{n'}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_k} : \mathbf{k}$$
$$\textit{where}\ \ell \mapsto \langle \mathbf{n}, \mathbf{k} \rangle \in \beta$$

5.2 Compiler Correctness

To state this, we need to tell when two heaps are related ($\mathsf{H} \approx_\beta \mathbf{H}$), that is when two β-related locations point to related values. Then, we need to tell when two values (and final values) are related ($\mathsf{v} \approx_\beta \mathbf{v}$).

$$\frac{}{\mathsf{fail} \approx_\beta \mathbf{fail}}\ (\text{Rel-fail}) \qquad \frac{}{\mathsf{true} \approx_\beta \mathbf{true}}\ (\text{Rel-val-true}) \qquad \frac{}{\mathsf{false} \approx_\beta \mathbf{false}}\ (\text{Rel-val-false}) \qquad \frac{}{\mathsf{n} \approx_\beta \mathbf{n}}\ (\text{Rel-val-n})$$

$$\frac{\ell \mapsto \langle \mathbf{n}, \mathbf{k} \rangle \in \beta}{\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle}\ (\text{Rel-val-loc}) \qquad \frac{\mathbf{H} \equiv \varnothing \text{ or } \mathbf{H} \equiv \mathbf{k_1}; \cdots; \mathbf{k_n}}{\varnothing \approx_\beta \mathbf{H}}\ (\text{Rel-heap-no})$$

$$\frac{\mathsf{H} \approx_\beta \mathbf{H} \qquad \ell \approx_\beta \langle \mathbf{n}', \mathbf{k} \rangle}{\mathsf{H}; \ell \mapsto \mathsf{n} \approx_\beta \mathbf{H}; \mathbf{n}' \mapsto \mathbf{n} : \mathbf{k}} \text{(Rel-heap)}$$

$$\frac{\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle \qquad \mathsf{H} \approx_\beta \mathbf{H} \qquad \mathsf{n}' \approx_\beta \mathbf{n}' \qquad \mathsf{H}' \approx_\beta \mathbf{H}'}{\mathsf{call}\ \ell\ \mathsf{H?} \cdot \mathsf{ret}\ \mathsf{n}'\ \mathsf{H}'! \approx_\beta \mathbf{call}\ \langle \mathbf{n}, \mathbf{v} \rangle\ \mathbf{H?} \cdot \mathbf{ret}\ \mathbf{n}'\ \mathbf{H}'!} \text{(Rel-trace)}$$

At this point, since we have added cases to the compiler, we need to add the missing cases to the compiler correctness proof and to the auxiliary lemmas. These additions are straightforward.

Theorem 5.1 (Compiler correctness for $[\![\cdot]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}}$).

$$\begin{aligned}
\forall \cdots \exists \beta' \supseteq \beta \ \textit{if}\ & \varnothing \vdash \mathsf{P} : \tau \\
\textit{and}\ & \vdash \mathsf{C} : \mathsf{Bool} \\
\textit{and}\ & \mathbf{C} = [\![\mathsf{C}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \\
\textit{and}\ & \mathbf{C}\left[[\![\mathsf{P}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}}\right] \hookrightarrow^* [\![\mathsf{H}, \beta']\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \triangleright [\![\mathsf{f}, \beta']\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \\
\textit{then}\ & \mathsf{C}[\mathsf{P}] \hookrightarrow^* \mathsf{H} \triangleright \mathsf{f} \\
& \mathsf{f} \approx_{\beta'} \mathbf{f} \\
& \mathsf{H} \approx_{\beta'} \mathbf{H}
\end{aligned}$$

Proof. By Lemma 5.2 (Expression correctness for $[\![\cdot]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}}$).

Lemma 5.2 (Expression correctness for $[\![\cdot]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}}$).

$$\begin{aligned}
\forall \cdots \exists \beta' \supseteq \beta \ \textit{if}\ & [\![\mathsf{H}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \triangleright [\![\mathsf{e}, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} [\![\gamma, \beta]\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \hookrightarrow^* [\![\mathsf{H}', \beta']\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \triangleright [\![\mathsf{f}, \beta']\!]^{\mathsf{S}_\ell}_{\mathbf{T}_\mathbf{k}} \\
& \mathsf{H} \approx_\beta \mathbf{H} \\
\textit{then}\ & \mathsf{H} \triangleright \mathsf{e}\gamma \hookrightarrow^* \mathsf{H}' \triangleright \mathsf{f} \\
& \mathsf{f} \approx_{\beta'} \mathbf{f} \\
& \mathsf{H}' \approx_{\beta'} \mathbf{H}'
\end{aligned}$$

Simple adaptation of the previous proof.

5.3 $\langle\!\langle\!\langle \cdot \rangle\!\rangle\!\rangle^{\mathbf{T}_\mathbf{k}}_{\mathsf{S}_\ell}$: Trace-based Backtranslation

In this case we have a single trace to backtranslate into a single source context.

$$\begin{aligned}
\langle\!\langle\!\langle \mathbf{call}\ \langle \mathbf{n}, \mathbf{v} \rangle\ \mathbf{H?} \cdot \mathbf{ret}\ \mathbf{n}'\ \mathbf{H}'! \rangle\!\rangle\!\rangle^{\mathbf{T}_\mathbf{k}}_{\mathsf{S}_\ell} = {} & \mathsf{let}\ \mathsf{y}{=}\mathsf{call}\ \mathsf{fn}\ \mathsf{e}\ \mathsf{in}\ \mathsf{e}' \\
& \text{where } \mathsf{e} = \ \mathsf{let}\ \mathsf{z}{=}\mathtt{allocate}\,(\mathbf{H}, \mathbf{n})\ \mathsf{in}\ \mathsf{z} \\
& \qquad \mathsf{e}' = \mathtt{update}\,(\mathbf{H}\ \mathit{from}\ \mathbf{H}')
\end{aligned}$$

$$\mathtt{allocate}\,(\mathbf{H}, \mathbf{n}) = 0 \qquad \text{if } \mathbf{n} \notin \mathtt{dom}\,(\mathbf{H})$$

$$
\begin{aligned}
&= \mathsf{x_n} && \text{if } \mathbf{H} = \varnothing \\
&= \text{let } \mathsf{x_n} = \text{new } \langle\!\langle\!\langle \mathbf{v} \rangle\!\rangle\!\rangle_{\mathsf{S}_\ell}^{\mathbf{T}_k} \text{ in} && \text{if } \mathbf{H} = \mathbf{H}'; \mathbf{n} \mapsto \mathbf{v} : \eta \\
&\quad \texttt{allocate}\,(\mathbf{H}') &&
\end{aligned}
$$

$$
\texttt{update}\,(\ \mathbf{H}\ \textit{from}\ \mathbf{H}') = \mathsf{true}
$$

When backtranslating the allocation, we stop if we see that the heap is not going to be used anyway, otherwise we allocate it and make sure that the correct variable is returned. Note that backtranslating the return is not necessary here, the compiled code will do that. However, we keep that structure as it will be useful later.

5.3.1 Properties of the Backtranslation

In order to use the context backtranslation, we need to prove that it is correct. We must define when behaviours are related ($\mathsf{b} \approx_\beta \mathbf{b}$), that is when their values and heaps are related.

$$
\frac{\ell \approx_\beta \langle \mathbf{n}, \mathbf{k} \rangle \qquad \mathsf{H} \approx_\beta \mathbf{H} \qquad \mathsf{n}' \approx_\beta \mathbf{n}' \qquad \mathsf{H}' \approx_\beta \mathbf{H}'}{\mathsf{call}\ \ell\ \mathsf{H?} \cdot \mathsf{ret}\ \mathsf{n}'\ \mathsf{H}'! \approx_\beta \mathbf{call}\ \langle \mathbf{n}, \mathbf{v} \rangle\ \mathbf{H?} \cdot \mathbf{ret}\ \mathbf{n}'\ \mathbf{H}'!}\ \text{(Rel-trace)}
$$

Additionally, we need to know that just before performing the "call", heaps are related.

Lemma 5.3 (Call correctness).

$$
\begin{aligned}
\textit{if}\quad & \mathsf{P} = \mathsf{H}_0 \triangleright \mathsf{fn}(\mathsf{x}) \mapsto \mathsf{e_f} \\
& \mathbf{C}\left[\llbracket \mathsf{P}, \beta \rrbracket_{\mathbf{T_u}}^{\mathsf{S}_\tau}\right] \hookrightarrow^* \mathbf{H} \triangleright \mathbf{let\ y = let\ x = \langle n, v \rangle\ in\ let\ x = read\ z.1\ with\ z.2\ in\ if\ x\ has\ Nat\ then} \llbracket \mathsf{e}, \beta \rrbracket_{\mathbf{T_k}}^{\mathsf{S}_\ell} \mathbf{\ else\ fail\ in\ e_c} \\
\textit{then}\ \exists \beta' \supseteq \beta\ & \langle\!\langle\!\langle \mathbf{call}\ \langle \mathbf{n}, \mathbf{v} \rangle\ \mathbf{H?} \cdot \mathbf{ret}\ \mathbf{n}'\ \mathbf{H}'! \rangle\!\rangle\!\rangle_{\mathsf{S}_\ell}^{\mathbf{T}_k} [\mathsf{P}] \hookrightarrow^* \mathsf{H} \triangleright \mathsf{let\ y = let\ x = } \ell \mathsf{\ in\ e_f\ in\ e_c} \\
& \mathsf{H} \approx_{\beta'} \mathbf{H} \\
& \ell \approx_{\beta'} \langle \mathbf{n}, \mathbf{v} \rangle
\end{aligned}
$$

Proof. Simple unfolding of definitions.

Expression correctness will then tell us that given related heaps and related arguments, a program and its compilation produce related outputs with related heaps. These two together will tell that the backtranslation is correct.

Theorem 5.4 (Correctness of the backtranslation of behaviours).

$$
\textit{if}\ \mathbf{b} \in \texttt{Behav}\left(\mathbf{C}\left[\llbracket \mathsf{P}, \beta \rrbracket_{\mathbf{T_u}}^{\mathsf{S}_\tau}\right]\right)
$$

$$\mathsf{b} \approx_\beta \mathbf{b}$$

$$\textit{then } \mathsf{b} \in \mathtt{Behav}\left(\langle\!\langle \mathbf{b} \rangle\!\rangle_{\mathsf{S}_\ell}^{\mathbf{T}_\mathbf{k}} [\mathsf{P}]\right)$$

Proofsketch. By Lemma 5.3 (Call correctness) and Lemma 5.2 (Expression correctness for $[\![\cdot]\!]_{\mathbf{T}_\mathbf{k}}^{\mathsf{S}_\ell}$).

5.3.2 Proving RSC

Theorem 5.5 (RSC for $[\![\cdot]\!]_{\mathbf{T}_\mathbf{k}}^{\mathsf{S}_\ell}$).

$$\forall \mathsf{P}, \mathbf{C}, \mathbf{b}, \exists \mathsf{C}, \beta, \mathsf{b} \approx_\beta \mathbf{b}.$$
$$\textit{if } \mathbf{C}\left[[\![\mathsf{P}, \beta]\!]_{\mathbf{T}_\mathbf{k}}^{\mathsf{S}_\ell}\right] \rightsquigarrow \mathbf{b}$$
$$\textit{then } \mathsf{C}[\mathsf{P}] \rightsquigarrow \mathsf{b}$$

Proof. This follows directly from Theorem 5.4 (Correctness of the backtranslation of behaviours).

6 Compiler Security: Robustly Safe Compilation

6.1 Target Memory Protection: Isolation $\mathbf{T}_\mathbf{e}$

6.1.1 Target Behaviour

6.1.2 Compiler Changes

7 Comparing Secure Compilation Proofs

7.1 Trace Difference

7.2 Proof Decomposition

7.3 Additional Backtranslation Steps

7.3.1 Masking

7.3.2 Maintaining a Local Heap Model

7.4 Discussion

A Appendix: Proofs

The appendix provides general indications on how the proofs proceed and omits some cases for students to practice proofs.

A.1 Proof of Lemma 1.1

Proof of Lemma 1.1. By structural induction over e.

Base case $\mathsf{e}{=}\mathsf{n}$

$\mathsf{e}{=}\mathsf{true}$

$\mathsf{e}{=}\mathsf{false}$

$\mathsf{e}{=}\mathsf{x}$

Inductive case In this case we identify these inductive hypotheses:

1. if $\mathsf{e}{\hookrightarrow^*}\mathsf{n}$ then $[\![\mathsf{e}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \hookrightarrow^* [\![\mathsf{n}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$;
2. if $\mathsf{e}'{\hookrightarrow^*}\mathsf{n}'$ then $[\![\mathsf{e}']\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \hookrightarrow^* [\![\mathsf{n}']\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$;
3. if $\mathsf{e}_\mathsf{b}{\hookrightarrow^*}\mathsf{v}$ then $[\![\mathsf{e}_\mathsf{b}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}} \hookrightarrow^* [\![\mathsf{v}]\!]^{\mathsf{S}_\tau}_{\mathbf{T_u}}$ (only in the "if" case).

The following cases arise:

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

$\mathsf{e}{=}\mathsf{if\ e_b\ then\ e\ else\ e'}$

$\mathsf{e}{=}\mathsf{e} \oplus \mathsf{e}'$

$\mathsf{e}{=}\mathsf{e} \odot \mathsf{e}'$

A.2 Proof of Lemma 3.8

Proof of Lemma 3.8. By structural induction over $\mathbf{e}$.

Base case $\mathbf{e}{=}\mathbf{n}$

$\mathbf{e}{=}\mathbf{true}$

$\mathbf{e}{=}\mathbf{false}$

$\mathbf{e}{=}\mathbf{fail}$

$\mathbf{e}{=}\mathbf{x}$

Inductive case In this case we identify these inductive hypotheses:

1. if $\mathbf{e}{\hookrightarrow^*}\mathbf{f}$ then $\langle\!\langle \mathbf{e} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \hookrightarrow^* \langle\!\langle \mathbf{f} \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$;
2. if $\mathbf{e}'{\hookrightarrow^*}\mathbf{f}'$ then $\langle\!\langle \mathbf{e}' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \hookrightarrow^* \langle\!\langle \mathbf{f}' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$;
3. if $\mathbf{e}''{\hookrightarrow^*}\mathbf{f}''$ then $\langle\!\langle \mathbf{e}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau} \hookrightarrow^* \langle\!\langle \mathbf{f}'' \rangle\!\rangle^{\mathbf{T_u}}_{\mathsf{S}_\tau}$ (only needed in the "if" case).

The following cases arise:

$\mathbf{e}$=$\mathbf{let\ x = e\ in\ e'}$

$\mathbf{e}$=$\mathbf{if\ e\ then\ e'\ else\ e''}$

$\mathbf{e}$=$\mathbf{e} \oplus \mathbf{e'}$

$\mathbf{e}$=$\mathbf{e} \odot \mathbf{e'}$

A.3 Proof of Lemma 3.12

Proof of Lemma 3.12. The proof proceeds by contradiction.

Assume the thesis is false: wlog we have: $\mathbf{P_1} \not\simeq_{ctx} \mathbf{P_2}$, so $\exists \mathbf{C}.\mathbf{C}[\mathbf{P_1}]\Downarrow$ *and* $\mathbf{C}[\mathbf{P_2}]\Uparrow$

Let us take a look at the traces of $\mathbf{C}[\mathbf{P_1}]$ and $\mathbf{C}[\mathbf{P_2}]$ respectively.

They are of the form $\mathbf{call\ n?} \cdot \mathbf{ret\ n_1!}$ and $\mathbf{call\ n?} \cdot \mathbf{ret\ n_2!}$.

By determinism of the semantics, $\mathbf{n}$ must coincide since it comes from the same $\mathbf{C}$.

By analysing the semantics, the only way for $\mathbf{C}$ to behave differently is to receive two different numbers $\mathbf{n_1}$ and $\mathbf{n_2}$.

This contradicts the assumption that traces are equal.