Secure Compilation Lecture Notes*

Marco Patrignani

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

Contents

1 Pure Whole Languages 4
  1.1 Source Language $S^w$ ........................................ 4
    1.1.1 Syntax .................................................. 4
    1.1.2 Dynamic Semantics ...................................... 4
    1.1.3 Static Semantics ........................................ 4
  1.2 Target Language $T^w$ ...................................... 5
    1.2.1 Syntax .................................................. 5
    1.2.2 Dynamic Semantics ..................................... 5
  1.3 $\llbracket^{S^w}_{T^w}$: Compiler from $S^w$ to $T^w$ ...... 6
  1.4 Compiler Correctness for $\llbracket^{S^w}_{T^w}$ ............ 6

2 Partial Languages 7
  2.1 Source Language Additions: $S_r$ .......................... 7
  2.2 Target Language Additions: $T_u$ .......................... 7
  2.3 Common Definitions ........................................ 7
  2.4 $\llbracket^{S_r}_{T_u}$: Compiler from $S_r$ to $T_u$ ...... 8
  2.5 Compiler Correctness for $\llbracket^{S_r}_{T_u}$ ............ 8

3 Compiler Security: Fully Abstract Compilation 8
  3.1 Compiler Changes ........................................... 9
  3.2 Fully Abstract Compilation via Context-based Backtranslation 9
    3.2.1 Reflection .............................................. 9
    3.2.2 Preservation ........................................... 9
  3.3 $\llbracket^{S_r}_{T_u}$: Context-based Backtranslation ... 10
    3.3.1 The Universal Type .................................... 10
    3.3.2 Helper Functions ...................................... 10
    3.3.3 Properties of the Helpers ............................. 10
    3.3.4 The Backtranslation ................................... 10
    3.3.5 Properties of the Backtranslation .................... 12
    3.3.6 Fully Abstract Compilation Reflection ............... 12
    3.3.7 Proper Differentiation ................................ 15
    3.3.8 Completing the Full Abstraction Proof ................ 16

*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 feedback and endurance: (sc101 @ Cispa&UdS’18), (cs350 @ Stanford’18), (scs19 @ Cispa&UdS’19), (cs350 @ Stanford’19)
3.4 Fully Abstract Compilation via Trace-based Backtranslation
  3.4.1 Target Changes: $T_r$ .......................... 16
  3.4.2 Target Trace Semantics .......................... 16
  3.4.3 Trace Equivalence .............................. 17
  3.4.4 Compiler from $S_\ell$ to $T_r$ .................. 17
3.5 $\langle\langle\cdot\rangle\rangle_{S_\ell}^{T_r}$: Trace-based Backtranslation .......................... 17
  3.5.1 Properties of the Backtranslation of Traces .......................... 17
  3.5.2 Fully Abstract Compilation .......................... 17
  3.5.3 Fully Abstract Compilation Preservation .......................... 18

4 Stateful Languages 18
  4.1 Source Heap: $S_\ell$ .............................. 18
    4.1.1 Syntax .................................. 18
    4.1.2 Dynamic Semantics ........................... 19
    4.1.3 Static Semantics ............................. 19
  4.2 Target Heap: $T_n$ ................................. 20
    4.2.1 Syntax .................................. 20
    4.2.2 Dynamic Semantics ........................... 20
  4.3 Common Definitions ................................ 20
    4.3.1 Behaviours ................................ 21
  4.4 $J_{\cdot}^K_{S_\ell}T_n$: Compiler from $S_\ell$ to $T_n$ .............. 21
  4.4.1 Compiler Shortcomings ........................... 22

5 Compiler Security: Robustly Safe Compilation 22
  5.1 Target Memory Protection: Capabilities $T_k$ ....................... 22
    5.1.1 Syntax .................................. 22
    5.1.2 Dynamic Semantics ........................... 23
    5.1.3 $\langle\langle\cdot\rangle\rangle_{S_\ell}^{T_k}$: Compiler from $S_\ell$ to $T_k$ .............. 23
  5.2 Compiler Correctness ................................ 24
  5.3 $\langle\langle\cdot\rangle\rangle_{S_\ell}^{T_k}$: Trace-based Backtranslation ..................... 25
    5.3.1 Properties of the Backtranslation .......................... 26
    5.3.2 Proving RSC ................................. 27

6 Compiler Security: Robustly Safe Compilation 27
  6.1 Target Memory Protection: Isolation $T_e$ ....................... 27
    6.1.1 Target Behaviour ................................ 27
    6.1.2 Compiler Changes ................................ 27

7 Comparing Secure Compilation Proofs 27
  7.1 Trace Difference ................................ 27
  7.2 Proof Decomposition ................................ 27
  7.3 Additional Backtranslation Steps .............................. 27
    7.3.1 Masking .................................. 27
    7.3.2 Maintaining a Local Heap Model ...................... 27
  7.4 Discussion .................................. 27
A Appendix: Proofs

A.1 Proof of Lemma 1.1 . . . . . . . . . . . . . . . . . . . . . . . . . 28
A.2 Proof of Lemma 3.8 . . . . . . . . . . . . . . . . . . . . . . . . 28
A.3 Proof of Lemma 3.12 . . . . . . . . . . . . . . . . . . . . . . . . 29
1 Pure Whole Languages

1.1 Source Language $S^w$

1.1.1 Syntax

\[
P ::= e \\
e ::= \text{true} \mid \text{false} \mid e \circ e \mid n \mid x \mid \text{let } x = e \text{ in } e \mid \text{if } e \text{ then } e \text{ else } e \mid \text{fail} \\
v ::= \text{true} \mid \text{false} \mid n \\
f ::= n \mid \text{fail} \\
E ::= [] \mid e \oplus E \mid E \oplus n \mid \text{let } x = E \text{ in } e \mid \text{if } E \text{ then } e \text{ else } e \mid e \odot E \mid E \odot n \\
\gamma ::= \emptyset \mid \gamma; [v / x]
\]

The program state is just an expression $e$.

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

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

1.1.2 Dynamic Semantics

Judgements: $e \hookrightarrow^0 e'$

\[
\begin{align*}
\text{(Eval-letin)} & \quad \text{let } x = n \text{ in } e \hookrightarrow^0 e[n / x] \\
\text{(Eval-op)} & \quad n \oplus n' = n'' \\
\text{(Eval-bop)} & \quad n \odot n' \hookrightarrow^0 n'' \\
\text{(Eval-if-t)} & \quad \text{if true then } e \text{ else } e' \hookrightarrow^0 e \\
\text{(Eval-if-f)} & \quad \text{if false then } e \text{ else } e' \hookrightarrow^0 e' \\
\text{(Eval-ctx)} & \quad E[e] \hookrightarrow E[e'] \\
\text{(Eval-fail)} & \quad E[\text{fail}] \hookrightarrow \text{fail}
\end{align*}
\]

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

1.1.3 Static Semantics

\[
\tau ::= \text{Bool} \mid \text{Nat} \mid \text{Nat} \to \text{Nat} \\
\Gamma ::= \emptyset \mid \Gamma; x : \tau
\]

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \text{Well-typed expression } e \text{ of type } \tau \\
\vdash \text{fn}(x) \mapsto e : \text{Nat} \to \text{Nat} & \quad \text{Well-typed program } e \text{ of type Nat to Nat}
\end{align*}
\]
\[ \Gamma \vdash \text{true} : \text{Bool} \]
\[ \Gamma \vdash \text{false} : \text{Bool} \]
\[ \Gamma \vdash n : \text{Nat} \]
\[ \Gamma \vdash x : \tau \in \Gamma \]
\[ \Gamma \vdash e_1 \circ e_2 : \text{Bool} \]
\[ \Gamma \vdash e_1 \circ e_2 : \text{Nat} \]
\[ \Gamma \vdash e_1 \circ e_2 : \tau \]
\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau \]
\[ \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \]
\[ \Gamma \vdash \text{fail} : \tau \]
\[ \Gamma \vdash \text{fn}(x) \mapsto e : \text{Nat} \rightarrow \text{Nat} \]

### 1.2 Target Language \( T^w \)

#### 1.2.1 Syntax

\[ P ::= e \]
\[ v ::= n \mid \text{true} \mid \text{false} \]
\[ b ::= \text{true} \mid \text{false} \]
\[ e ::= n \mid x \mid e \circ e \mid \text{let } x = e \text{ in } e \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid \text{fail} \]
\[ T ::= \text{Nat} \mid \text{Bool} \]
\[ f ::= n \mid \text{fail} \]
\[ E ::= [\cdot] \mid e \circ E \mid E \circ v \mid \text{let } x = E \text{ in } e \mid \text{if } E \text{ then } e_1 \text{ else } e_2 \mid E \circ v \]
\[ \gamma ::= \emptyset \mid \gamma : [v / x] \]

#### 1.2.2 Dynamic Semantics

**Judgements:**

\[ e \mapsto^0 e' \quad e \mapsto^0 e' \quad e \mapsto^0 e' \]

\[ \text{(Eval-letin)} \]
\[ \text{let } x = v \text{ in } e \mapsto^0 e[v / x] \]

\[ \text{(Eval-op)} \]
\[ n \circ n' = n'' \quad n \circ n' = v \]

\[ \text{(Eval-bop)} \]
\[ n \circ n' \mapsto^0 v \]

\[ \text{(Eval-if-t)} \]
\[ \text{if true then e else e'} \mapsto^0 e \]

\[ \text{(Eval-if-f)} \]
\[ \text{if false then e else e'} \mapsto^0 e' \]

\[ \text{(Eval-fail)} \]
\[ E[\text{fail}] \mapsto \text{fail} \]

\[ \text{(Eval-has-nN)} \]
\[ n \text{ has N} \mapsto^0 \text{true} \]

\[ \text{(Eval-has-vB)} \]
\[ v \equiv \text{true or false} \quad v \text{ has N} \mapsto^0 \text{false} \]

\[ \text{(Eval-has0nB)} \]
\[ n \text{ has B} \mapsto^0 \text{false} \]

\[ \text{(Eval-cxt)} \]
\[ e \mapsto^0 e' \]

\[ \text{(Eval-has0N)} \]
\[ v \equiv \text{true or false} \quad v \text{ has N} \mapsto^0 \text{true} \]
\[ v = b \text{ or } v' = b \]
\[ v \oplus v' \rightarrow \text{fail} \]
\[ (\text{Eval-op-fail}) \]

\[ v = b \text{ or } v' = b \]
\[ v \odot v' \rightarrow \text{fail} \]
\[ (\text{Eval-bop-fail}) \]

\[ \text{if } n \text{ then } e \text{ else } e' \rightarrow \text{fail} \]
\[ (\text{Eval-if-fail}) \]

1.3 \( \llbracket \cdot \rrbracket_{T^w} : \text{Compiler from } S^w \text{ to } T^w \)

\[ \llbracket \text{true} \rrbracket_{T^w} = \text{true} \]
\[ \llbracket \text{false} \rrbracket_{T^w} = \text{false} \]
\[ \llbracket n \rrbracket_{T^w} = n \]
\[ \llbracket x \rrbracket_{T^w} = x \]
\[ \llbracket e \oplus e' \rrbracket_{T^w} = \llbracket e \rrbracket_{T^w} \oplus \llbracket e' \rrbracket_{T^w} \]
\[ \llbracket e \odot e' \rrbracket_{T^w} = \llbracket e \rrbracket_{T^w} \odot \llbracket e' \rrbracket_{T^w} \]
\[ \llbracket \text{let } x = e \text{ in } e' \rrbracket_{T^w} = \text{let } x = \llbracket e \rrbracket_{T^w} \text{ in } \llbracket e' \rrbracket_{T^w} \]
\[ \llbracket \text{if } b \text{ then } e \text{ else } e' \rrbracket_{T^w} = \text{if } \llbracket b \rrbracket_{T^w} \text{ then } \llbracket e \rrbracket_{T^w} \text{ else } \llbracket e' \rrbracket_{T^w} \]
\[ \llbracket \text{fail} \rrbracket_{T^w} = \text{fail} \]
\[ \llbracket \llbracket v \rrbracket_{T^w} / x \rrbracket_{T^w} = \llbracket \llbracket v \rrbracket_{T^w} \rrbracket_{T^w} / \llbracket x \rrbracket_{T^w} \rrbracket \]

1.4 Compiler Correctness for \( \llbracket \cdot \rrbracket_{T^w} \)

Lemma 1.1 (Forward simulation).

\[ \text{if } e_\gamma \rightarrow^* v \text{ then } \llbracket e \rrbracket_{T^w} \llbracket \gamma \rrbracket_{T^w} \rightarrow^* \llbracket v \rrbracket_{T^w} \]

Find the proof template in Appendix A.1.

Lemma 1.2 (Expression correctness).

\[ \text{if } \llbracket e \rrbracket_{T^w} \llbracket \gamma \rrbracket_{T^w} \rightarrow^* \llbracket f \rrbracket_{T^w} \text{ then } e_\gamma \rightarrow^* f \]

Proof. By contradiction we assume: \( e_\gamma \rightarrow v' \neq v \) (the case for \( b \) is analogous).

By Lemma 1.1 (Forward simulation) we get that \( \llbracket e_\gamma \rrbracket_{T^w} \llbracket \gamma \rrbracket_{T^w} \rightarrow^* \llbracket v \rrbracket_{T^w} \).

By determinism of the compiler we have \( \llbracket v \rrbracket_{T^w} \neq \llbracket v' \rrbracket_{T^w} \).

So we have that the same term \( \llbracket e \rrbracket_{T^w} \) reduces to two different terms, which contradicts the determinism of the semantics. \( \square \)

Theorem 1.3 (Compiler correctness for \( \llbracket \cdot \rrbracket_{T^w} \) (Whole Programs)).

\[ \text{if } \emptyset \vdash P : \tau \text{ and } \llbracket P \rrbracket_{T^w} \rightarrow^* \llbracket f \rrbracket_{T^w} \text{ then } P \rightarrow^* f \]
Proof. By Lemma 1.2 (Expression correctness).

2 Partial Languages

2.1 Source Language Additions: $S_r$

\[
P ::= \text{fn}(x) \mapsto e \\
C ::= \text{let } y = \text{call fn e in e}
\]

\[\vdash \text{let } y = \text{call fn e}_1 \text{ in } e_2 : \text{Bool} \quad \text{Well-typed context of type } \text{Bool}\]

\[
\begin{array}{c}
\vdash e_1 : \text{Nat} \\
y : \text{Nat} \vdash e_2 : \text{Bool} \\
\vdash \text{let } y = \text{call fn e}_1 \text{ in } e_2 : \text{Bool}
\end{array}
\]

2.2 Target Language Additions: $T_u$

\[
P ::= \text{fn}(x) \mapsto e \\
C ::= \text{let } y = \text{call fn e}
\]

2.3 Common Definitions

Definition 2.1 (Contextual equivalence).

\[
P_1 \sim_{ctx} P_2 \overset{\text{def}}{=} \forall C. \ C[P_1] \mapsto^* f \text{ and } C[P_2] \mapsto^* f
\]

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

\[
C[P] \overset{\text{def}}{=} \text{let } y = \text{let } x = e_{\text{arg in e}_{\text{fun in e}_{\text{cont}}}}
\]

Example 2.3 (Equivalent and inequivalent programs).

\[
\begin{align*}
\text{x + 2} & \quad \text{x + 1 + 1} \\
\text{if x > 0 then 0 else 1} & \quad \text{x} \\
\text{let z = x in z + z} & \quad \text{z + z}
\end{align*}
\]
2.4 \([\cdot]_{T_u}^{S_r}: \text{Compiler from } S_r \text{ to } T_u\)

\[\begin{align*}
[\text{fn}(x) \mapsto e]_{T_u}^{S_r} &= \text{fn}(x) \mapsto [e]_{T_u}^{S_r} \\
[e]_{T_u}^{S_r} &= [e]_{Tw}^{S_r} \\
[\text{let } y = \text{call fn e in } e]_{T_u}^{S_r} &= \text{let } y = \text{call fn } [e]_{T_u}^{S_r} \text{ in } [e]_{T_u}^{S_r}
\end{align*}\]

2.5 Compiler Correctness for \([\cdot]_{T_u}^{S_r}\)

Theorem 2.4 (Compiler correctness for \([\cdot]_{T_u}^{S_r}\) (Partial Programs)).

\[\begin{align*}
\text{if } \emptyset \vdash P : \tau \text{ and } \vdash C : \text{Bool and } C = [C]_{Tw}^{S_w} \text{ and } C \left[ [P]_{Tw}^{S_w} \right] &\leadsto^* [f]_{Tw}^{S_w} \\
\text{then } C \left[ P \right] &\leadsto^* f
\end{align*}\]

Proof. Analogous to the proof of Theorem 1.3 (Compiler correctness for \([\cdot]_{Tw}^{S_w}\) (Whole Programs)).

3 Compiler Security: Fully Abstract Compilation

Theorem 3.1 (Full abstraction of \([\cdot]_{T_u}^{S_r}\)).

\[\forall P_1, P_2. \quad P_1 \simeq_{ctx} P_2 \iff [P_1]_{T_u}^{S_r} \simeq_{ctx} [P_2]_{T_u}^{S_r}\]

Example 3.2 (Violations of full abstraction).

\[\begin{align*}
C_b &= \text{let } y = \text{call fn true in } 2 \\
P_1 &= \text{fn}(x) \mapsto \text{let } z = x \text{ in } 1 \\
P_2 &= \text{fn}(x) \mapsto 1
\end{align*}\]

In this case, we have \(\forall C, \exists f:\)

- \(C \left[ P_1 \right] \leadsto^* f\)
- \(C \left[ P_2 \right] \leadsto^* f\)
- \(C_b \left[ [P_1]_{T_u}^{S_w} \right] \leadsto^* \text{fail}\)
- \(C_b \left[ [P_2]_{T_u}^{S_w} \right] \leadsto^* 2\)

So, \(P_1 \simeq_{ctx} P_2\) but \([P_1]_{Tw}^{S_w} \not\simeq_{ctx} [P_2]_{Tw}^{S_w}\). \(\square\)
3.1 Compiler Changes

\[ \text{fn}(x) \mapsto e \mapsto \text{if } x \text{ has } N \text{ then } e \mapsto \text{else fail} \]

3.2 Fully Abstract Compilation via Context-based Back-translation

3.2.1 Reflection

Lemma 3.3 (Equivalence reflection for \([\cdot]_{T_u}^S\).

\[ \forall P_1, P_2. \ P_1 \simeq_{ctx} P_2 \iff [P_1]_{T_u}^S \simeq_{ctx} [P_2]_{T_u}^S \]

Proof. We state this in contrapositive form:

\[ P_1 \not\simeq_{ctx} P_2 \Rightarrow [P_1]_{T_u}^S \not\simeq_{ctx} [P_2]_{T_u}^S \]

and expand the definitions of \(\simeq_{ctx}\):

\[ \exists C. \ C[P_1] \mapsto_f^* \text{ and } C[P_2] \mapsto_{f'}^* \text{ and } f \neq f' \]

\[ \Rightarrow \exists C. \ C[[P_1]_{T_u}^S] \mapsto_f^* \text{ and } C[[P_2]_{T_u}^S] \mapsto_{f'}^* \text{ and } f \neq f' \]

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

3.2.2 Preservation

Lemma 3.4 (Equivalence preservation for \([\cdot]_{T_u}^S\).

\[ \forall P_1, P_2. \ P_1 \simeq_{ctx} P_2 \Rightarrow [P_1]_{T_u}^S \simeq_{ctx} [P_2]_{T_u}^S \]

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

\[ \forall P_1, P_2. \ [P_1]_{T_u}^S \not\simeq_{ctx} [P_2]_{T_u}^S \Rightarrow P_1 \not\simeq_{ctx} P_2 \]

and expand the definitions of \(\simeq_{ctx}\):

\[ \forall P_1, P_2. \ \exists C. \ C[[P_1]_{T_u}^S] \mapsto_f^* \text{ and } C[[P_2]_{T_u}^S] \mapsto_{f'}^* \text{ and } f \neq f' \]

\[ \Rightarrow \exists C. \ C[P_1] \mapsto_f^* \text{ and } C[P_2] \mapsto_{f'}^* \text{ and } f \neq f' \]

We can try to build \(C\) starting from \(C\) as we cannot rely on any correctness result.
3.3 $\langle \cdot \rangle_{S_{T_u}}^T$: 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 true to true because when backtranslating $3 + \text{true}$ we would get $3 + \text{true}$ 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., $3 \ast 2$ 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 call f true to 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.

\[
\begin{align*}
\text{inject}_{\text{Nat}}(e) &= e + 2 \\
\text{inject}_{\text{Bool}}(e) &= \text{if } e \text{ then } 1 \text{ else } 0 \\
\text{extract}_{\text{Nat}}(e) &= \text{let } x = e \text{ in if } x \geq 2 \text{ then } x - 2 \text{ else } \text{fail} \\
\text{extract}_{\text{Bool}}(e) &= \text{let } x = e \text{ in if } x \geq 2 \text{ then fail else if } x - 1 \geq 1 \text{ then false else true}
\end{align*}
\]

3.3.3 Properties of the Helpers

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

- If $\Gamma \vdash e : \text{Nat}$ then $\Gamma \vdash \text{inject}_{\text{Nat}}(e) : \text{Nat}$
- If $\Gamma \vdash e : \text{Bool}$ then $\Gamma \vdash \text{inject}_{\text{Bool}}(e) : \text{Nat}$
- If $\Gamma \vdash e : \text{Nat}$ then $\Gamma \vdash \text{extract}_{\text{Nat}}(e) : \text{Nat}$
- If $\Gamma \vdash e : \text{Nat}$ then $\Gamma \vdash \text{extract}_{\text{Bool}}(e) : \text{Bool}$

Proof. Simple case analysis.

3.3.4 The Backtranslation

The backtranslation is based on the program context structure.

\[
\langle \text{let } y = \text{call } fn \ e' \text{ in } e'' \rangle_{S_{T_u}}^T = \text{let } y = \text{inject}_{\text{Nat}}(\text{call } fn (\text{extract}_{\text{Nat}}(\langle e' \rangle_{S_{T_u}}^T))) \text{ in } \langle e'' \rangle_{S_{T_u}}^T
\]
\[\langle n \rangle_T = n + 2\]
\[\langle x \rangle_T = x\]
\[\langle \text{true} \rangle_T = 0\]
\[\langle \text{false} \rangle_T = 1\]
\[\langle e \oplus e' \rangle_T = \text{let } x_1 = \text{extract}_{\text{Nat}} \langle e \rangle_T \text{ in} \]
\[\text{let } x_2 = \text{extract}_{\text{Nat}} \langle e' \rangle_T \text{ in} \]
\[\text{inject}_{\text{Nat}} x_1 \oplus x_2\]
\[\langle e \odot e' \rangle_T = \text{let } x_1 = \text{extract}_{\text{Nat}} \langle e \rangle_T \text{ in} \]
\[\text{let } x_2 = \text{extract}_{\text{Nat}} \langle e' \rangle_T \text{ in} \]
\[\text{inject}_{\text{Bool}} x_1 \oplus x_2\]
\[\langle \text{let } x = e \text{ in } e' \rangle_T = \langle e \rangle_T \text{ in} \langle e' \rangle_T\]
\[\langle \text{if } e \text{ then } e' \text{ else } e'' \rangle_T = \text{let } x = \langle e \rangle_T \text{ in} \]
\[\text{if } x \geq 2 \text{ then } \langle e' \rangle_T \text{ else } \langle e'' \rangle_T\]
\[\langle \text{e has } T \rangle_T = \begin{cases} 
\text{let } x = \langle e \rangle_T \text{ in if } x \geq 2 \text{ then } 1 \text{ else } 0 & \text{if } T = \text{Nat} \\
\text{let } x = \langle e \rangle_T \text{ in if } x \geq 2 \text{ then } 0 \text{ else } 1 & \text{if } T = \text{Bool}
\end{cases}\]
\[\langle \text{fail} \rangle_T = \text{fail}\]
\[\langle [v / x] \rangle_T = \left[ \langle v \rangle_T / \langle x \rangle_T \right]\]

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

\textbf{Remark 3.7 (Letin).} The backtranslation of \text{let } x = e \text{ in } e' may seem confusing, as it does not insert \text{inject}_{\text{Nat}} for its subexpressions.

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

- \(e_1 = \text{let } x = 2 \text{ in } x + 1\) and \(t_1 \rightarrow^* 3\);

Currently, what we get is:

- \(e_1 = \text{let } x = 4 \text{ in } \text{let } x_1 = \text{extract}_{\text{Nat}} x \text{ in let } x_2 = \text{extract}_{\text{Nat}} 3 \text{ in } \text{inject}_{\text{Nat}} x_1 + x_2\)

If we unfold the reductions, we see that
let x = 4 in let x1 = \text{extract}_{\mathbb{Nat}} x in let x2 = \text{extract}_{\mathbb{Nat}} 3 \implies inject_{\mathbb{Nat}} x1 + x2
\begin{align*}
e_1 & \implies^* let x2 = \text{extract}_{\mathbb{Nat}} 3 \implies inject_{\mathbb{Nat}} 2 + 1 \\
& \implies^* 5
\end{align*}
and these reductions proceed as expected.

However if we insert an additional \text{extract}_{\mathbb{Nat}} for the value bound to the x, these reductions will not go right, as we get an additional \text{+2}. We could eliminate it by adding an \text{inject}_{\mathbb{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:

- \begin{align*}e_2 &= \text{let } x = \text{true} \text{ in if } x \text{ then 3 else 0}\end{align*}

This gets backtranslated to

- \begin{align*}e_2 &= \text{let } x = 0 \text{ in let } z = \text{extract}_{\mathbb{Bool}} x \text{ in if } z == 1 \text{ then 5 else 2}\end{align*}

These expressions reduce correctly, but we would not know how to carry the information that x is technically a Boolean (the \text{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 \text{\oplus} expression, the \text{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).**

\[
\text{if } e_{\gamma} \implies^* f \Rightarrow \langle\langle e_{\gamma}\rangle\rangle_{S_{\tau}T_u} \implies^* \langle\langle f\rangle\rangle_{S_{\tau}T_u}
\]

Find the proof template in Appendix A.2.

### 3.3.6 Fully Abstract Compilation Reflection

We resume our proof for the ⇒ direction of fully abstract compilation.

**Proof of Lemma 3.4.** What we have is:

\[
\exists C. C \left[\llbracket P_1 \rrbracket_{T_u}^{S} \right] \implies^* f \text{ and } C \left[\llbracket P_2 \rrbracket_{T_u}^{S} \right] \implies^* f' \text{ and } f \neq f'
\Rightarrow \exists C. C[P_1] \implies^* f' \text{ and } C[P_2] \implies^* f' \text{ and } f \neq f'
\]

We can instantiate C with \langle\langle C\rangle\rangle_{S_{\tau}T_u}.

So we can assume:
1. $C \left[ [P_1]^{S_T}_{T_u} \right] \hookrightarrow^+ f$

2. $C \left[ [P_2]^{S_T}_{T_u} \right] \hookrightarrow^+ f'$

and prove this

- $\langle C \rangle^{T_u}_{S_T} [P_1] \hookrightarrow^+ f$
- $\langle C \rangle^{T_u}_{S_T} [P_2] \hookrightarrow^+ f'$

If we unfold the definition of $C \left[ P \right]$ in the hypotheses, assuming that $P_1 = \text{fn}(x) \mapsto e_1$ and $P_2 = \text{fn}(x) \mapsto e_2$ we obtain:

1. let $y = \text{let } x = e' \text{ in if } x \text{ has NAT then } [e_1]^{S_T}_{T_u} \text{ else fail in } e'' \hookrightarrow^+ f$

2. let $y = \text{let } x = e' \text{ in if } x \text{ has NAT then } [e_2]^{S_T}_{T_u} \text{ else fail in } e'' \hookrightarrow^+ f'$

We can unfold the reductions to see that:

- $\text{let } y = \text{let } x = e' \text{ in if } x \text{ has NAT then } [e_1]^{S_T}_{T_u} \text{ else fail in } e''$ 
  $\hookrightarrow^* \text{let } y = \text{let } x = v' \text{ in if } x \text{ has NAT then } [e_1]^{S_T}_{T_u} \text{ else fail in } e''$
  $\text{let } y = \text{if } v' \text{ has NAT then } ([e_1]^{S_T}_{T_u}[v'/x]) \text{ else fail in } e''$

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

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:

- $\text{let } y = \text{let } x = e' \text{ in if } x \text{ has NAT then } [e_2]^{S_T}_{T_u} \text{ else fail in } e''$
  $\hookrightarrow^* \text{let } y = \text{if } v' \text{ has NAT then } ([e_2]^{S_T}_{T_u}[v'/x]) \text{ else fail in } e''$

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

So $v'$ really is a natural number $n'$:

- $\text{let } y = \text{if } n' \text{ has NAT then } ([e_1]^{S_T}_{T_u}[n'/x]) \text{ else fail in } e''$
  $\hookrightarrow^* \text{let } y = ([e_1]^{S_T}_{T_u}[n'/x]) \text{ in } e''$
  $\text{let } y = ([n_1]^{S_T}_{T_u}) \text{ in } e''$
  $\hookrightarrow^* e''[\langle n_1 \rangle^{S_T}_{T_u} / y]$
  $\hookrightarrow^* f$

The execution of $P_2$, instead, must differ, so, determinism of the semantics let us conclude that these reductions happen, for a $n_2 \neq n_1$:

- $\text{let } y = \text{if } n' \text{ has NAT then } ([e_2]^{S_T}_{T_u}[n'/x]) \text{ else fail in } e''$
  $\hookrightarrow^* e''[\langle n_2 \rangle^{S_T}_{T_u} / y]$
  $\hookrightarrow^* f'$
Let us take a look at the source reductions. By Lemma 3.8 (Backtranslation correctness), we know the following:

\[
\begin{align*}
P_1 & \quad \text{let } y = \text{inject}_{\text{Nat}}(\text{let } x = \text{extract}_{\text{Nat}}(\langle\langle e'\rangle\rangle)_{\mathcal{T}_u} \text{ in } e_1) \text{ in } \langle\langle e''\rangle\rangle_{\mathcal{T}_u} \\
\quad \quad \rightarrow^* \text{ let } y = \text{inject}_{\text{Nat}}(\text{let } x = \text{extract}_{\text{Nat}}(\langle\langle v'\rangle\rangle)_{\mathcal{T}_u} \text{ in } e_1) \text{ in } \langle\langle e''\rangle\rangle_{\mathcal{T}_u}
\end{align*}
\]

By inspecting the target reductions we know, \(\text{extract}_{\text{Nat}}(\langle\langle v'\rangle\rangle)_{\mathcal{T}_u}\) cannot fail, as \(v'\) is a natural number. Additionally, we know that \(v' = \langle\langle n'\rangle\rangle_{\mathcal{T}_u} - 2\). By Lemma 1.2 (Expression correctness) we know:

\[
\begin{align*}
P_1 & \quad \text{let } y = \text{inject}_{\text{Nat}}(\langle\langle e''\rangle\rangle)_{\mathcal{T}_u} \\
\quad \quad \rightarrow^* \text{ let } y = \text{inject}_{\text{Nat}}n_1 \text{ in } \langle\langle e''\rangle\rangle_{\mathcal{T}_u} \\
\quad \quad \text{So by definition of } \text{inject}_{\text{Nat}} \text{ we have that } n_1' = n_1 + 2:
\end{align*}
\]

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

\[
\begin{align*}
P_2 & \quad \text{let } y = \text{inject}_{\text{Nat}}(\text{let } x = \text{extract}_{\text{Nat}}(\langle\langle e''\rangle\rangle)_{\mathcal{T}_u} \text{ in } e_2) \text{ in } \langle\langle e''\rangle\rangle_{\mathcal{T}_u} \\
\quad \quad \rightarrow^* \text{ let } y = \text{inject}_{\text{Nat}}n_2 \text{ in } \langle\langle e''\rangle\rangle_{\mathcal{T}_u} \\
\quad \quad \text{We can define } n_2' = n_2 + 2 \text{ and we get:}
\end{align*}
\]

At this stage in the proof we make a simplification for the sake of clarity.

For simplicity, we can assume that \(e''\) is if \(y = [n_1]_{\mathcal{T}_u}\) then 0 else 1 and thus \(f = 0\) and \(f' = 1\).

Given that we know \(e''\), let us work out its backtranslation (modulo some optimisation and elimination of bits that we know how will reduce):
let \( z = \text{extract}_{\text{Bool}} \langle \langle y == n_1 \rangle \rangle \}_{S^*} \) in if \( z == 1 \) then \( \langle \langle 0 \rangle \rangle \) else \( \langle \langle 1 \rangle \rangle \)

\[
\begin{align*}
&= \text{let } x1 = \text{extract}_{\text{Nat}} \langle \langle y \rangle \rangle \}_{S^*} \text{ in } \\
&\quad \text{let } x2 = \text{extract}_{\text{Nat}} \langle \langle n_1 \rangle \rangle \}_{S^*} \text{ in } \\
&\quad \text{inject}_{\text{Bool}} x1 == x2 \\
&= \text{let } x = \text{extract}_{\text{Bool}} \text{ let } x1 = y - 2 \text{ in } \\
&\quad \text{let } x2 = n_1 \text{ in } \\
&\quad \text{inject}_{\text{Bool}} x1 == x2 \\
&= \text{if } y - 2 == n_1 \text{ then } 2 \text{ else } 3
\end{align*}
\]

So we have

\[
\begin{align*}
P_1 & \quad \text{if } y - 2 == n_1 \text{ then } 2 \text{ else } 3[n_1' / y] \\
& = \text{if } n_1' - 2 == n_1 \text{ then } 2 \text{ else } 3 \\
& = \text{if } n_1 + 2 - 2 == n_1 \text{ then } 2 \text{ else } 3 \\
& \hookrightarrow 2
\end{align*}
\]

Now we know \( \langle \langle e'' \rangle \rangle_{S^*} \) and we know that \( n_2 \neq n_1 \), so we have that

\[
\begin{align*}
P_2 & \quad = \text{if } y - 2 == n_1 \text{ then } 2 \text{ else } 3[n_2' / y] \\
& = \text{if } n_2 + 2 - 2 == n_1 \text{ then } 2 \text{ else } 3 \\
& = \text{if } n_2 == n_1 \text{ then } 2 \text{ else } 3 \\
& \hookrightarrow 3
\end{align*}
\]

So \( f = 2 \) and \( f' = 3 \) and this proof holds. \( \square \)

### 3.3.7 Proper Differentiation

Note that the assumption on what \( e'' \) 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).**

if \( f \neq f' \) then \( \langle \langle f \rangle \rangle_{S^*} \neq \langle \langle f' \rangle \rangle_{S^*} \)

**Proof.** Trivial case analysis.

**Proper completion of Lemma 3.4.** We have:

- \( e'' \left[ n_1 \right]_{T^*}^S / y \mapsto^* f' \)
- \( e'' \left[ n_2 \right]_{T^*}^S / y \mapsto^* f' \)

and we need to reason about
Since \( n'_1 = n_1 + 2 \), \( n'_2 = n_2 + 2 \), we can rewrite the last two items as
\[
\langle\langle e'\rangle\rangle_{\mathcal{T}_r}[\langle\langle n_1 \rangle\rangle_{\mathcal{S}_r} \rightarrow \langle\langle y \rangle\rangle_{\mathcal{S}_r}]
\]
\[
\langle\langle e'\rangle\rangle_{\mathcal{T}_r}[\langle\langle n_2 \rangle\rangle_{\mathcal{S}_r} \rightarrow \langle\langle y \rangle\rangle_{\mathcal{S}_r}]
\]
By how backtranslation works on substitutions, we can apply Lemma 3.8 (Back-
translation correctness) twice to conclude that:
\[
\langle\langle e'\rangle\rangle_{\mathcal{T}_r}[\langle\langle n_1 \rangle\rangle_{\mathcal{S}_r} \rightarrow \langle\langle y \rangle\rangle_{\mathcal{S}_r}]
\]
\[
\langle\langle e'\rangle\rangle_{\mathcal{T}_r}[\langle\langle n_2 \rangle\rangle_{\mathcal{S}_r} \rightarrow \langle\langle y \rangle\rangle_{\mathcal{S}_r}]
\]
Finally, we apply Lemma 3.9 (Differentiation) to conclude
\[
\langle\langle f \rangle\rangle_{\mathcal{T}_r} \neq \langle\langle f' \rangle\rangle_{\mathcal{T}_r}
\]
3.4.3 Trace Equivalence

Definition 3.11 (Trace equivalence).
\[ P_1 \equiv_{T} P_2 \text{ def } = TR(P_1) = TR(P_2) \]

Lemma 3.12 (Soundness of traces).
\[ P_1 \equiv_{T} P_2 \Rightarrow P_1 \simeq_{ctx} P_2 \]

Find the proof in Appendix A.3.

3.4.4 Compiler from $S_r$ to $T_r$

\[ \llbracket P \rrbracket_{S_r}^{T_r} = \llbracket P \rrbracket_{T_u}^{S_r} \]

3.5 $\langle\langle\langle \cdot \rangle\rangle\rangle_{S_r}$: 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.

\[ t_1 = \text{call } n? \cdot \text{ret } n^1! \quad \quad \quad t_2 = \text{call } n? \cdot \text{ret } n^2! \]

\[ \langle\langle\langle n \rangle\rangle\rangle_{S_r}^{T_r} = n \]

\[ \langle\langle\langle t_1, t_2 \rangle\rangle\rangle_{S_r}^{T_r} = \text{let } y = \text{call fn } \langle\langle\langle n \rangle\rangle\rangle_{S_r}^{T_r} \text{ in if } y == \langle\langle\langle n^1 \rangle\rangle\rangle_{S_r}^{T_r} \text{ then 1 else 2} \]

3.5.1 Properties of the Backtranslation of Traces

Lemma 3.13 (Correctness of the backtranslation of traces).

\[
\begin{align*}
\text{if } \text{call } n? \cdot \text{ret } n^1! \in TR(\llbracket P \rrbracket_{T_u}^{S_r}) \\
P = \text{fn}(x) \mapsto e \\
\text{then } \text{let } y = \text{let } x = \langle\langle\langle n \rangle\rangle\rangle_{S_r}^{T_r} \text{ in e in e' } \mapsto^* e' \left[ \langle\langle\langle n' \rangle\rangle\rangle_{S_r}^{T_r} / y \right]
\end{align*}
\]

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

3.5.2 Fully Abstract Compilation

Theorem 3.14 (Full abstraction of $\llbracket \cdot \rrbracket_{T_r}^{S_r}$).

\[ \forall P_1, P_2. \ P_1 \simeq_{ctx} P_2 \iff \llbracket P_1 \rrbracket_{T_r}^{S_r} \simeq_{ctx} \llbracket P_2 \rrbracket_{T_r}^{S_r} \]

Proof. Since the compiler has not changed, the reflection holds by Lemma 3.3 (Equivalence reflection for $\llbracket \cdot \rrbracket_{T_u}^{S_r}$). The preservation holds by Lemma 3.15 (Equivalence preservation for $\llbracket \cdot \rrbracket_{T_r}^{S_r}$) below. \(\square\)
3.5.3 Fully Abstract Compilation Preservation

Lemma 3.15 (Equivalence preservation for $J \cdot K$).

$$\forall P_1, P_2. \ P_1 \simeq_{ctx} P_2 \Rightarrow [[P_1]]^{S_{\tau}}_{T_r} \simeq_{ctx} [[P_2]]^{S_{\tau}}_{T_r}$$

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

$$\forall P_1, P_2. \ P_1 \simeq_{ctx} P_2 \Rightarrow [[P_1]]^{S_{\tau}}_{T_r} T = [[P_2]]^{S_{\tau}}_{T_r}$$

in contrapositive form:

$$\forall P_1, P_2. \ [[P_1]]^{S_{\tau}}_{T_r} T \neq [[P_2]]^{S_{\tau}}_{T_r} \Rightarrow P_1 \not\simeq_{ctx} P_2$$

expanding the definitions:

$$\forall P_1, P_2. \ \exists t \in TR ([[P_1]]^{S_{\tau}}_{T_r}) \text{ and } t \not\in TR ([[P_2]]^{S_{\tau}}_{T_r})$$

$$\Rightarrow \exists C. \ C[P_1] \rightarrow \text{"n" and } C[P_2] \rightarrow \text{"n"}$$

We pick another trace $t'$ from $TR ([[P_2]]^{S_{\tau}}_{T_r})$ such that the first part is the same as in $t$ for the backtranslation. Knowing that $[[n_1]]^{S_{\tau}}_{T_r} = n^1$ and $n^1 \not= n^2$, the traces are

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

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

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

$$[[P_1]]^{S_{\tau}}_{T_r} \quad \begin{array}{l}
\rightarrow^* \text{if } y == \langle\langle\langle n^1 \rangle\rangle\rangle^{S_{\tau}_{T_r}} \text{ then } 1 \text{ else } 2 \\
\equiv \text{if } n_1 == \langle\langle\langle n^1 \rangle\rangle\rangle^{S_{\tau}_{T_r}} \text{ then } 1 \text{ else } 2
\end{array}$$

and

$$[[P_2]]^{S_{\tau}}_{T_r} \quad \begin{array}{l}
\rightarrow^* \text{if } y == \langle\langle\langle n^2 \rangle\rangle\rangle^{S_{\tau}_{T_r}} \text{ then } 1 \text{ else } 2 \\
\equiv \text{if } n_2 == 1 == \langle\langle\langle n^2 \rangle\rangle\rangle^{S_{\tau}_{T_r}} \text{ then } 1 \text{ else } 2
\end{array}$$

The different reductions now are straightforward, so $f = 1$ and $f' = 2$ so this case holds.

□

4 Stateful Languages

4.1 Source Heap: $S_\ell$

4.1.1 Syntax

$$P ::= H \triangleright \text{fn}(x) \rightarrow e$$
\[ e ::= \cdots | \text{let } x = \text{new } e \text{ in } e | \text{let } x = \text{read } e \text{ in } e | \text{let } x = \text{write } e \text{ at } e \text{ in } e \]
\[ v ::= v | \ell \]
\[ E ::= \cdots | \text{let } x = \text{new } E \text{ in } e | \text{let } x = \text{read } E \text{ in } e | \text{let } x = \text{write } E \text{ at } e \text{ in } e \]
\[ H ::= \emptyset | H ; \ell \mapsto n \]
\[ \Omega ::= H \]

Programs have a global heap.
\( H(\ell) \) returns the number \( n \) in \( H \) to which \( \ell \) maps to.
\( H \cup \ell \mapsto n \) updates \( H \) with a (possibly new) binding of the form \( \ell \mapsto n \). If \( \ell \) is in the domain of \( H \), the binding for \( \ell \) is updated to point to \( n \).

### 4.1.2 Dynamic Semantics

Judgement: \( \Omega \xrightarrow{\cdot} \Omega' \)

\[
\begin{align*}
&\text{(Eval-ctx)} & H \triangleright e \xrightarrow{0} H \triangleright e' & &\text{(Eval-new)} & \ell \notin H & H \triangleright \text{let } x = \text{new } n \text{ in } e \xrightarrow{0} H \cup \ell \mapsto n \triangleright e' / \ell \\
& & H \triangleright e \triangleright H \triangleright e' & &\text{(Eval-read)} & H \triangleright \text{let } x = \text{read } \ell \text{ in } e \xrightarrow{0} H \triangleright e[H(\ell)] / x \\
& & & &\text{(Eval-write)} & H \triangleright \text{let } x = \text{write } n \text{ at } \ell \text{ in } e'' \xrightarrow{0} H \cup \ell \mapsto n \triangleright e''[n / x] \\
\end{align*}
\]

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.

\[ \tau ::= \cdots | \text{Ref Nat} \]
\[ \Sigma ::= \emptyset | \Sigma ; \ell : \text{Ref Nat} \]
\[ \vdash H \triangleright \text{fn}(x) \mapsto e : \text{Ref Nat} \rightarrow \text{Nat} \quad \text{Well-typed program } e \text{ of type Ref Nat to Nat} \]

\[
\begin{align*}
&\text{(T-alloc)} & \Gamma, \Sigma \vdash e : \text{Nat} & \Gamma ; x : \text{Ref Nat}, \Sigma \vdash e' : \tau & \Gamma, \Sigma \vdash \text{let } x = \text{new } e \text{ in } e' : \tau \\
&\text{(T-read)} & \Gamma, \Sigma \vdash e : \text{Ref Nat} & \Gamma ; x : \text{Nat}, \Sigma \vdash e' : \tau & \Gamma, \Sigma \vdash \text{let } x = \text{read } e \text{ in } e' : \tau
\end{align*}
\]
Γ, Σ ⊢ e : Nat  Γ, Σ ⊢ e′ : Ref Nat  Γ; x : Nat, Σ ⊢ e′′ : τ
Γ, Σ ⊢ let x = write e at e′ in e′′ : τ

(Π-loc)
ℓ ∈ dom(Σ)
Γ, Σ ⊢ ℓ : Ref Nat

(Π-prog)
x : Nat, Σ ⊢ e : Nat
fail ∉ e
Σ = dom(H) : Ref Nat
Γ ⊢ H ⊩ fn(x) ↦ e : Ref Nat → Nat

4.2 Target Heap: T_n

4.2.1 Syntax

P ::= H ⊩ fn(x) ↦ e

E ::= · · · | let x = new e in e | let x = read e in e | let x = write e at e in e

H ::= · · · | H; n ↦→ v

Ω ::= H ⊩ e

4.2.2 Dynamic Semantics

Judgement: Ω ⊩ Ω′

(Eval-ctx)
H ⊩ e ↦→ 0 H ⊩ e′
H ⊩ e ↦→ H ⊩ e′

the cardinality of H = n

H ⊩ let x = new v in e ↦→ H ⊩ n + 1 ↦→ v ⊩ e[x / n + 1]

(Eval-read)
n ⊩ v ∈ H

H ⊩ let x = read n in e ↦→ H ⊩ v[ x / x]

(Eval-read-no)
n ∉ dom(H)

H ⊩ let x = read n in e ↦→ H ⊩ e[false / x]

(Eval-write)
n ⊩ v ∈ H

H ⊩ let x = write v at n in e′′ ↦→ H ⊩ n ⊩ v′ ∼ e′′[n / x]

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

4.3 Common Definitions

Assume both languages also contain pairs (e, e) and projections e.1, e.2 as standardly done.
Also, we will use a shorthand $e_1;e_2$ for the expression \( \text{let } _ = e_1 \ \text{in} \ e_2 \).

**Definition 4.1 (Plugging).**

\[
\begin{align*}
\text{if } C &= \text{let } y = \text{call fn e in } e_c \\
\text{and } P &= H \triangleright \text{fn} (x) \mapsto e_{\text{fun}}
\end{align*}
\]

then $C [P] \overset{\text{df}}{=} H \triangleright \text{let } y = \text{let } x = e \ \text{in} \ e_{\text{fun}} \ \text{in} \ e_c$

### 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.

\[ b ::= \text{call } n \ H? \cdot \text{ret } n \ H! \]

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

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

\[
\text{Behav} (C[P]) \overset{\text{df}}{=} \{ b \mid C[P] \rightsquigarrow b \}
\]

### 4.4 $S^{S_\ell}_{T_n}$: Compiler from $S_\ell$ to $T_n$

The compiler maps a source location to a target number.

\[
\begin{align*}
[H \triangleright \text{fn} (x) \mapsto e]_{T_n}^{S_\ell} &= [H]_{T_n}^{S_\ell} \triangleright \text{fn} (z) \mapsto \text{let } x = \text{read } z \ \text{in} \\
&\quad \text{if } x \text{ has Nat then } [e, H]_{T_n}^{S_\ell} \text{ else fail} \\
[\text{let } x = \text{new } e \ \text{in} \ e', H]_{T_n}^{S_\ell} &= \text{let } x_1 = \text{new } [e, H]_{T_n}^{S_\ell} \ \text{in} \ [e', H]_{T_n}^{S_\ell} \\
[\text{let } x = \text{read } e \ \text{in} \ e', H]_{T_n}^{S_\ell} &= \text{let } [x]_{T_n}^{S_\ell} = \text{read } [e, H]_{T_n}^{S_\ell} \ \text{in} \ [e', H]_{T_n}^{S_\ell} \\
[\text{let } x = \text{write } e \ \text{at} e' \ \text{in} \ e'', H]_{T_n}^{S_\ell} &= \text{let } [x]_{T_n}^{S_\ell} = \text{write } [e, H]_{T_n}^{S_\ell} \ \text{at} \ [e', H]_{T_n}^{S_\ell} \ \text{in} \ [e'', H]_{T_n}^{S_\ell}
\end{align*}
\]
\[
\llbracket \ell, H \rrbracket_{T_n}^{S_i} = n
\]
where \( \text{indexOf}(\ell, H) = n \)

\[
\llbracket \emptyset \rrbracket_{T_n}^{S_i} = \emptyset
\]
\[
\llbracket H; \ell \mapsto n \rrbracket_{T_n}^{S_i} = \llbracket H \rrbracket_{T_n}^{S_i}; n + 1 \mapsto \llbracket n', \emptyset \rrbracket_{T_n}^{S_i}
\]
where \( \text{card}(H) = n \)

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.

\[
\begin{align*}
P_1 &= \ell \mapsto \ell + 2 \triangleright \text{fn}(x) \mapsto \text{let } y = \text{read } \ell \text{ in } y + 1; 3 \\
P_2 &= \ell \mapsto \ell + 2 \triangleright \text{fn}(x) \mapsto 3 \\
C_b &= \text{let } y = \text{call fn let } z = \text{write true at } 1 \text{ in } z \text{ in true}
\end{align*}
\]

Obviously \( P_1 \simeq_{ctx} P_2 \).

The reductions of \( \llbracket P_1 \rrbracket_{T_n}^{S_i} \) will get stuck when doing \( \llbracket y + 1 \rrbracket_{T_n}^{S_i} \) since \( \llbracket y \rrbracket_{T_n}^{S_i} \) will contain the \text{true} value written there by \( C_b \). On the other hand, the reductions of \( \llbracket P_2 \rrbracket_{T_n}^{S_i} \) will not get stuck, so \( \llbracket P_1 \rrbracket_{T_n}^{S_i} \not\simeq_{ctx} \llbracket P_2 \rrbracket_{T_n}^{S_i} \).

### 5 Compiler Security: Robustly Safe Compilation

#### 5.1 Target Memory Protection: Capabilities \( T_k \)

In this case we do not extend the language \( T_n \), but the language \( T_u \) into the new language \( T_k \).

##### 5.1.1 Syntax

\[
\begin{align*}
v &::= \cdots | k \\
e &::= \cdots | \text{let } x = \text{new } e \text{ in } e \ | \text{let } x = \text{read } e \text{ with } e \text{ in } e \\
&\quad | \text{let } x = \text{write } e \text{ at } e \text{ with } e \text{ in } e \ | \text{let } x = \text{hide } e \text{ in } e \\
c &::= \text{let } y = \text{call fn } e \text{ in } e \text{ such that let } x = \text{hide } e \text{ in } e \notin e \\
e &::= \cdots | \text{let } x = \text{new } E \text{ in } e \ | \text{let } x = \text{read } E \text{ with } e \text{ in } e \\
&\quad | \text{let } x = \text{write } E \text{ at } e \text{ with } e \text{ in } e \ | \text{let } x = \text{write } v \text{ at } E \text{ with } e \text{ in } e \\
&\quad | \text{let } x = \text{hide } E \text{ in } e \\
h &::= \emptyset | H; n \mapsto n : \eta | H; k
\end{align*}
\]
\( \eta ::= \bot \mid k \)
\( \Omega ::= H \triangleright e \)

### 5.1.2 Dynamic Semantics

Judgement: \( \Omega \triangleright \Omega' \)

- (Eval-new)
  \( H \triangleright \text{let } x = \text{new } n' \text{ in } e \triangleright H \cup n + 1 \triangleright n' \triangleright e[x / n + 1] \)
- (Eval-read)
  \( n \triangleright v : k \in H \)
  \( H \triangleright \text{let } x = \text{read } n \text{ with } k \text{ in } e \triangleright e[v / x] \)
- (Eval-read-no-cap)
  \( n \not\in \text{dom}(H) \)
  \( H \triangleright \text{let } x = \text{read } n \text{ with } v \text{ in } e \triangleright e[v / x] \)
- (Eval-read-no)
  \( n \not\in \text{dom}(H) \)
  \( H \triangleright \text{let } x = \text{read } n \text{ with } v \text{ in } e \triangleright e[\text{false} / x] \)
- (Eval-write)
  \( n \triangleright v : k \in H \)
  \( H \triangleright \text{let } x = \text{write } n' \text{ at } n \text{ with } k \text{ in } e'' \triangleright H \cup n \triangleright n' \triangleright e''[n / x] \)
- (Eval-write-no-cap)
  \( n \triangleright_\_ : \bot \in H \)
  \( H \triangleright \text{let } x = \text{write } n' \text{ at } n \text{ with } v \text{ in } e'' \triangleright H \cup n \triangleright n' \triangleright e''[n / x] \)

### 5.1.3 \([S_\ell]^{S_k}_{T_k} : Compiler from S_\ell to T_k\)

The compiler translates source locations \( \ell \) into pairs \( n, k \) of the natural number \( n \) which is the target location and a capability \( k \) used to hide location \( n \).

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, \( x \) will not pass the dynamic typecheck. Also, note that in the case of programs, the compiler remaps \( x \) as \( z \) in order to perform the typecheck on inputs.

\( \beta ::= \emptyset \mid \beta ; \ell \mapsto (n, k) \)

- \( \text{getbeta}(\emptyset) = \emptyset \)
- \( \text{getbeta}(H; \ell \mapsto n') = \text{getbeta}(H); \ell \mapsto (n + 1, k) \)

where \( \text{card}(H) = n \) and \( k \not\in \text{getbeta}(H).\text{capabilities} \)
Function \texttt{getbeta}(H) inputs a heap and returns a partial bijection \(\beta\) where all locations in the heap are bound to increasing numbers and fresh capabilities.

\[
[H \triangleright \text{fn}(x) \mapsto e, \beta]_T^S_k = [H, \beta]_T^S_k \triangleright \text{fn}(z) \mapsto \text{let } x = \text{read } z.1 \text{ with } z.2 \text{ in }
\]
\[
\text{if } x \text{ has Nat then } [e, \beta]_T^S_k \text{ else fail}
\]

\(\text{and } \beta = \text{getbeta}(H)\)

\[
\ldots
\]

\[
[\text{let } x = \text{new } e \text{ in } e', \beta]_T^S_k = \text{let } x_1 = \text{new } [e, \beta]_T^S_k \text{ in }
\]
\[
\text{let } x_c = \text{hide } x_1 \text{ in }
\]
\[
\text{let } [x, \beta]_T^S_k = (x_1, x_c) \text{ in }
\]
\[
[e', \beta]_T^S_k
\]

\[
[\text{let } x = \text{read } e \text{ in } e', \beta]_T^S_k = \text{let } x_p = [e, \beta]_T^S_k \text{ in }
\]
\[
\text{let } [x, \beta]_T^S_k = \text{read } x_p.1 \text{ with } x_p.2 \text{ in }
\]
\[
[e', \beta]_T^S_k
\]

\[
[\text{let } x = \text{write } e \text{ at } e' \text{ in } e'', \beta]_T^S_k = \text{let } x_p = [e', \beta]_T^S_k \text{ in }
\]
\[
\text{let } [x, \beta]_T^S_k = \text{write } [e, \beta]_T^S_k \text{ at } x_p.1 \text{ with } x_p.2 \text{ in }
\]
\[
[e', \beta]_T^S_k
\]
\[
[\ell, \beta]_T^S_k = \beta(\ell)
\]
\[
[\emptyset; \beta]_T^S_k = \emptyset
\]
\[
[H; \ell \mapsto n', \beta]_T^S_k = [H, \beta]_T^S_k ; n \mapsto [n', \beta]_T^S_k : k
\]

where \(\ell \mapsto (n, k) \in \beta\)

5.2 Compiler Correctness

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

\[
\begin{array}{cccc}
\text{(Rel-fail)} & \text{(Rel-val-true)} & \text{(Rel-val-false)} & \text{(Rel-val-n)} \\
\text{fail} \approx_\beta \text{fail} & \text{true} \approx_\beta \text{true} & \text{false} \approx_\beta \text{false} & \text{n} \approx_\beta \text{n} \\
\text{(Rel-val-loc)} & \ell \mapsto (n, k) \in \beta & \ell \approx_\beta (n, k) & \\
\text{(Rel-heap-no)} & H \equiv \emptyset \text{ or } H \equiv k_1; \cdots ; k_n & \emptyset \equiv_\beta H &
\end{array}
\]
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 \([\llbracket T_k S \rrbracket]\).**

\[
\forall \cdots \exists \beta' \supseteq \beta \text{ if } \emptyset \vdash P : \tau \\
\text{and } \vdash C : \text{Bool} \\
\text{and } C = \llbracket C, \beta \rrbracket_{T_k}^{S_k} \\
\text{and } C \llbracket [P, \beta]^{S_k} \rrbracket_{T_k} \leftarrow^* \llbracket [H, \beta']^{S_k} \rrbracket_{T_k} \leftarrow^* \llbracket [f, \beta']^{S_k} \rrbracket_{T_k} \\
\text{then } C[P] \leftarrow^* H \triangleright f \\
f \approx_{\beta'} f \\
H \approx_{\beta'} H
\]

*Proof.* By Lemma 5.2 (Expression correctness for \([\llbracket T_k S \rrbracket]\).

**Lemma 5.2 (Expression correctness for \([\llbracket T_k S \rrbracket]\).**

\[
\forall \cdots \exists \beta' \supseteq \beta \text{ if } \llbracket H, \beta \rrbracket_{T_k}^{S_k} \triangleright e \llbracket e, \beta' \rrbracket_{T_k}^{S_k} \leftarrow^* \llbracket H', \beta' \rrbracket_{T_k}^{S_k} \leftarrow^* \llbracket f, \beta' \rrbracket_{T_k}^{S_k} \\
H \approx_{\beta'} H \\
\text{then } H \triangleright e \triangleright^* H' \triangleright f \\
f \approx_{\beta'} f \\
H' \approx_{\beta'} H'
\]

Simple adaptation of the previous proof.

### 5.3 \(\llbracket \cdot \rrbracket_{T_k}^{S_k}\): Trace-based Backtranslation

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

\[
\llbracket \text{call } \langle n, v \rangle \ H? \cdot \text{ret } n' \ H'! \rrbracket_{T_k}^{S_k} = \text{let } y=\text{call } \text{fn } e \text{ in } e' \\
\text{where } e = \text{let } z=\text{allocate } (H, n) \text{ in } z \\
e' = \text{update } (H \text{ from } H')
\]

\(\text{allocate } (H, n) = 0 \text{ if } n \notin \text{dom}(H)\)
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 \((b \approx_{\beta} b')\), that is when their values and heaps are related.

\[
\begin{align*}
\ell \approx_{\beta} \langle n, k \rangle & \quad H \approx_{\beta} H & \quad n' \approx_{\beta} n' & \quad H' \approx_{\beta} H' \\
\text{call } \ell \ H? \cdot \text{ret } n' \ H'! & \approx_{\beta} \text{call } \langle n, v \rangle \ H? \cdot \text{ret } n' \ H'!
\end{align*}
\]

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

**Lemma 5.3 (Call correctness).**

\[
\text{if } \quad P = H_0 \triangleright \text{fn}(x) \mapsto e_r \\
C \left[ \left[ P, \beta \right]_{S_{\tau}}^{T_k} \right] \mapsto H \triangleright \text{let } y = \text{let } x = \langle n, v \rangle \text{ in } \\
\quad \text{let } x = \text{read } z.1 \text{ with } z.2 \text{ in } \\
\quad \text{if } x \text{ has Nat then } \left[ e, \beta \right]_{S_{\tau}}^{T_k} \text{ else fail } \\
\quad \text{in } e_c \\
\text{then } \exists \beta' \supseteq \beta \quad \langle \langle \text{call } \langle n, v \rangle \ H? \cdot \text{ret } n' \ H'! \rangle \rangle_{S_{\tau}}^{T_k} \left[ P \right] \mapsto H \triangleright \text{let } y = \text{let } x = \ell \text{ in } e_r \text{ in } e_c \\
H \approx_{\beta} H \\
\ell \approx_{\beta'} \langle n, v \rangle
\]

**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).**

\[
\text{if } b \in \text{Behav} \left( C \left[ \left[ P, \beta \right]_{S_{\tau}}^{T_k} \right] \right)
\]
\[ b \approx_\beta b \]
\[ \text{then } b \in \text{Behav}\left(\langle\langle\langle b \rangle\rangle\rangle_{T_k}^S |P|\right) \]

Proofsketch. By Lemma 5.3 (Call correctness) and Lemma 5.2 (Expression correctness for \([\parallel\parallel]_{T_k}^S\)).

5.3.2 Proving RSC

Theorem 5.5 (RSC for \([\parallel\parallel]_{T_k}^S\)).

\[ \forall P, C, b, \exists C, \beta, b \approx_\beta b. \]
\[ \text{if } C[\parallel P, \beta\rangle_{T_k}^S] \rightsquigarrow b \]
\[ \text{then } C[P] \rightsquigarrow b \]

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 \(T_e\)

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** \( e = n \)

- \( e = \text{true} \)
- \( e = \text{false} \)
- \( e = x \)

**Inductive case** In this case we identify these inductive hypotheses:

1. if \( e \rightarrow^n \) then \( [e]^{S_r}_{T_u} \rightarrow^n [n]^{S_r}_{T_u} \);
2. if \( e' \rightarrow^n \) then \( [e']^{S_r}_{T_u} \rightarrow^n [n]^{S_r}_{T_u} \);
3. if \( e_b \rightarrow^\ast \) then \( [e_b]^{S_r}_{T_u} \rightarrow^\ast [v]^{S_r}_{T_u} \) (only in the “if” case).

The following cases arise:

- \( e = \text{let } x = e \text{ in } e \)
- \( e = \text{if } e_b \text{ then } e \text{ else } e' \)
- \( e = e \oplus e' \)
- \( e = e \odot e' \)

A.2 Proof of Lemma 3.8

Proof of Lemma 3.8. By structural induction over \( e \).

**Base case** \( e = n \)

- \( e = \text{true} \)
- \( e = \text{false} \)
- \( e = \text{fail} \)
- \( e = x \)

**Inductive case** In this case we identify these inductive hypotheses:

1. if \( e \rightarrow^f \) then \( \langle \langle e \rangle \rangle^{S_r}_{T_u} \rightarrow^f \langle \langle f \rangle \rangle^{S_r}_{T_u} \);
2. if \( e' \rightarrow^f \) then \( \langle \langle e' \rangle \rangle^{S_r}_{T_u} \rightarrow^f \langle \langle f \rangle \rangle^{S_r}_{T_u} \);
3. if \( e'' \rightarrow^f \) then \( \langle \langle e'' \rangle \rangle^{S_r}_{T_u} \rightarrow^f \langle \langle f'' \rangle \rangle^{S_r}_{T_u} \) (only needed in the “if” case).
The following cases arise:

\[ e = \text{let } x = e \text{ in } e' \]
\[ e = \text{if } e \text{ then } e' \text{ else } e'' \]
\[ e = e \oplus e' \]
\[ e = e \odot e' \]

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: \( P_1 \not\cong P_2 \), so \( \exists C. C[P_1] \downarrow \text{ and } C[P_2] \uparrow \)

Let us take a look at the traces of \( C[P_1] \) and \( C[P_2] \) respectively.

They are of the form \( \text{call } n? \cdot \text{ret } n_1! \) and \( \text{call } n? \cdot \text{ret } n_2! \).

By determinism of the semantics, \( n \) must coincide since it comes from the same \( C \).

By analysing the semantics, the only way for \( C \) to behave differently is to receive two different numbers \( n_1 \) and \( n_2 \).

This contradicts the assumption that traces are equal.