Lecture 6: Proofs

Secure Compilation Seminar

Marco Patrignani
Why Proofs?

• large systems
• require a lot of time
• building and planning focus on two different aspects
• proofs ensure that the building is doable
Why Proofs?

• large systems
• require a lot of time
• building and planning focus on two different aspects
• proofs ensure that the building is doable (also why we have design patterns for coding)
How to Prove?

\[ P \Rightarrow Q \]
How to Prove?

\[ P \implies Q \]

- IF we can assume something \((P)\)
How to Prove?

$P \Rightarrow Q$

- IF we can assume something ($P$)
- THEN some other thing holds ($Q$)
Reduction ad Absurdum (or contradiction)

\[ P \Rightarrow Q \]

- assume \( P \)
- assume \( \neg Q \)
- derive \( \bot \) i.e., any contradiction (\( R \) and \( \neg R \))
Induction

\[ P \Rightarrow Q \]

• assume \( P \), prove \( Q(0) \) (base case)
Induction

\[ P \Rightarrow Q \]

- assume \( P \), prove \( Q(0) \) (base case)
- assume \( P \) and \( Q(n) \), prove \( Q(n + 1) \)
Induction

\[ P \implies Q \]

- assume \( P \), prove \( Q(0) \) (base case)
- assume \( P \) and \( Q(n) \), prove \( Q(n+1) \)
- generally \( Q \) has an infinite universal quantification
Structural Induction

\[ P \rightarrow Q \]

- generally done when \( Q \) has a (finite) structure
- e.g., reduction cases, typing cases, syntax
Contrapositive

\[ P \Rightarrow Q \]

becomes

\[ \neg Q \Rightarrow \neg P \]

and becomes oftentimes easier
What do we Prove?

• What are $P$ and $Q$?
What do we Prove?

• What are $P$ and $Q$?

$[\cdot]^S_T$ is FAC $\stackrel{\text{def}}{=} \forall P_1, P_2$

$P_1 \simeq_{ctx} P_2 \iff [P_1]^S_T \simeq_{ctx} [P_2]^S_T$
• break the $\iff$:
  1. $\Rightarrow$: $\forall P_1, P_2. \mathcal{P}_1 \approx_{ctx} \mathcal{P}_2 \Rightarrow [\mathcal{P}_1]^S_T \approx_{ctx} [\mathcal{P}_2]^S_T$
  2. $\Leftarrow$: $\forall P_1, P_2. [\mathcal{P}_1]^S_T \approx_{ctx} [\mathcal{P}_2]^S_T \Rightarrow \mathcal{P}_1 \approx_{ctx} \mathcal{P}_2$

• point 2 (should) follow from compiler correctness
Fully Abstract Compilation

- break the $\iff$:
  1. $\Rightarrow$: $\forall P_1, P_2. \ P_1 \sim_{ctx} P_2 \Rightarrow \llbracket P_1 \rrbracket_T^S \sim_{ctx} \llbracket P_2 \rrbracket_T^S$
  2. $\Leftarrow$: $\forall P_1, P_2. \ [P_1]^S_T \sim_{ctx} [P_2]^S_T \Rightarrow P_1 \sim_{ctx} P_2$

- point 2 (should) follow from compiler correctness

- point 1 is tricky, because of $\sim_{ctx}$ and its $\forall$
Trace Semantics

• we replace $\sim_{ctx}$ with something equivalent
Trace Semantics

- we replace \( \approx_{ctx} \) with something equivalent
- but simpler to reason about
Trace Semantics

• we replace $\simeq_{ctx}$ with something equivalent
• but simpler to reason about
• a semantics that abstracts from the context (observer)
• we replace $\simeq_{ctx}$ with something equivalent
• but simpler to reason about
• a semantics that abstracts from the context (observer)
• and still describes the behaviour of a program precisely
Trace Semantics

- we replace $\approx_{ctx}$ with something equivalent
- but simpler to reason about
- a semantics that abstracts from the context (observer)
- and still describes the behaviour of a program precisely
- a trace semantics
Traces for PMA

0x0001  call func. at 0xb52
0x0002  write r0 at 0x0b55

0xb52 write r0 at 0x0b55
0xb53 write r0 at 0x0001
0xb54 call 0x0002
0xb55 ...

0xab00 jump to 0x0001
0xab01 return to 0x0b53
0xab02 ...

- interest in the behaviour of the module
### Traces for PMA

<table>
<thead>
<tr>
<th>Address</th>
<th>Action</th>
<th>Address</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>call func. at 0xb52</td>
<td>0x0002</td>
<td>write r0 at 0x0b55</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xb52</td>
<td>write r0 at 0x0b55</td>
<td>0xb53</td>
<td>write r0 at 0x0001</td>
</tr>
<tr>
<td>0xb54</td>
<td>call 0x0002</td>
<td>0xb55</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ab00</td>
<td>jump to 0x0001</td>
<td>ab01</td>
<td>return to 0x0b53</td>
</tr>
<tr>
<td>ab02</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- interest in the behaviour of the module
- need to consider the rest
### Traces for PMA

<table>
<thead>
<tr>
<th>Address</th>
<th>Action</th>
<th>Address</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00001</td>
<td>call func. at 0xb52</td>
<td>0x00002</td>
<td>write r₀ at 0x0b55</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0b52</td>
<td>write r₀ at 0x0b55</td>
<td>0x0b53</td>
<td>write r₀ at 0x0001</td>
</tr>
<tr>
<td>0x0b54</td>
<td>call 0x0002</td>
<td>0x0b55</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xab00</td>
<td>jump to 0x0001</td>
<td>0xab01</td>
<td>return to 0x0b53</td>
</tr>
<tr>
<td>0xab02</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- interest in the behaviour of the module
- need to consider the *rest*
Trace Semantics for PMA

- disregard the rest

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00001</td>
<td>call func. at 0xb52</td>
<td>0x0002</td>
<td>write r₀ at 0x0b55</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0b52</td>
<td>write r₀ at 0x0b55</td>
<td>0x0b53</td>
<td>write r₀ at 0x0001</td>
</tr>
<tr>
<td>0x0b54</td>
<td>call 0x0002</td>
<td>0x0b55</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xab00</td>
<td>jump to 0x0001</td>
<td>0xab01</td>
<td>return to 0x0b53</td>
</tr>
<tr>
<td>0xab02</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trace Semantics for PMA

- disregard the rest
Trace Semantics for PMA

- disregard the rest
- abstract its behaviour from the module perspective:
Call args.

- disregard the rest
- abstract its behaviour from the module perspective:
  1. jump to an entry point

\[
\begin{array}{ll}
0xb52 & \text{write } r_0 \text{ at } 0xb55 \\
0xb53 & \text{write } r_0 \text{ at } 0x0001 \\
0xb54 & \text{call } 0x0002 \\
0xb55 & \ldots \\
\vdots & \\
0xab00 & \text{jump to } 0x0001 \\
0xab01 & \text{return to } 0xb53 \\
0xab02 & \ldots \\
\end{array}
\]
Trace Semantics for PMA

- disregard the rest
- abstract its behaviour from the module perspective:
  1. jump to an entry point
- abstract the module behaviour from the rest perspective:
Trace Semantics for PMA

- disregard the rest
- abstract its behaviour from the module perspective:
  1. jump to an entry point
- abstract the module behaviour from the rest perspective:
  1. call/return outside
Trace Semantics for PMA

- disregard the rest
- abstract its behaviour from the module perspective:
  1. jump to an entry point
- abstract the module behaviour from the rest perspective:
  1. call/return outside
  2. read/write
Trace Semantics

- semantics for partial programs (component)
Trace Semantics

- semantics for **partial programs** (component)
- relies on the operational semantics
Trace Semantics

- semantics for partial programs (component)
- relies on the operational semantics
- denotational: describes the behaviour of a component as sets of traces
Trace Semantics

• semantics for **partial programs** (component)
• relies on the operational semantics
• denotational: describes the **behaviour** of a component as **sets of traces**
• a **trace** is (typically) a sequence of **actions** that describe how a component interacts with an observer
Trace Semantics

• semantics for partial programs (component)
• relies on the operational semantics
• denotational: describes the behaviour of a component as sets of traces
• a trace is (typically) a sequence of actions that describe how a component interacts with an observer
• without needing to specify the observer
Trace Semantics

- semantics for **partial programs** (component)
- relies on the operational semantics
- denotational: describes the **behaviour** of a component as **sets of traces**
- a **trace** is (typically) a sequence of **actions** that describe how a component interacts with an observer
- **without** needing to specify the observer
- indicated as \( \text{TR}(C) = \left\{ \overline{\alpha} \mid C \xrightarrow{\overline{\alpha}} - \right\} \)
Trace Actions

Labels \( L ::= a \mid \epsilon \)

Observable actions \( \alpha ::= \sqrt{\mid} g? \mid g! \)

Actions \( g ::= \text{call } p(r) \mid \text{ret } p = r(r_0) \)
Traces for PMA

We need to define:

• trace states (almost program states) $\Theta$
• labels that make traces
• rules for generating labels and traces $\cdots$
• the traces of a component $\text{TR}(C) = \cdots$
Trace Equivalence

• all semantics yield a notion of equivalence
• all semantics yield a notion of equivalence
• the operational semantics gives us contextual equivalence

\[ C_1 \sim_{ctx} C_2 \]
• all semantics yield a notion of equivalence
• the operational semantics gives us contextual equivalence

\[ C_1 \approx_{ctx} C_2 \]

• trace semantics gives us trace equivalence

\[ C_1 \equiv T_2 \]
Trace Equivalence

• all semantics yield a notion of equivalence
• the operational semantics gives us contextual equivalence

\[ C_1 \sim_{\text{ctx}} C_2 \]

• trace semantics gives us trace equivalence

\[ \text{TR}(C_1) = \text{TR}(C_2) \]

the traces of \( C_1 \) are the same of those of \( C_2 \)
Trace Equivalence

- all semantics yield a notion of equivalence
- the operational semantics gives us contextual equivalence

\[ C_1 \simeq_{ctx} C_2 \]

- trace semantics gives us trace equivalence

\[ \{ \overline{\alpha} \mid C_1 \xrightarrow{\overline{\alpha}} - \} = \{ \overline{\alpha} \mid C_2 \xrightarrow{\overline{\alpha}} - \} \]

the traces of \( C_1 \) are the same of those of \( C_2 \)
• any trace semantics won’t just work
• they need to be correct and complete
Proofs about Trace Semantics

- any trace semantics won’t just work
- they need to be correct and complete

\[ C_1 \simeq_{ctx} C_2 \iff C_1 \equiv C_2 \]
Proofs about Trace Semantics

• any trace semantics won’t just work
• they need to be correct \( (\Leftarrow) \) and complete \( (\Rightarrow) \)

\[
C_1 \simeq_{ctx} C_2 \iff C_1 \models C_2
\]
• we have:
  • $C_1 \cong_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
• we have:
  • \( C_1 \sim_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \)

• we need to prove
  • \( P_1 \sim_{ctx} P_2 \Rightarrow [P_1]^S_T \sim_{ctx} [P_2]^S_T \)
we have:

- $C_1 \simeq_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2)$

we need to prove

- $P_1 \simeq_{ctx} P_2 \Rightarrow \forall C. \text{C}[\lbrack C_1 \rbrack^S_T] \downarrow \text{C}[\lbrack C_2 \rbrack^S_T]$

unfold $\simeq_{ctx}$
we have:

\[ C_1 \sim_{ctx} C_2 \iff TR(C_1) = TR(C_2) \]

we need to prove

\[ \exists C. \ C \left[ \left[ C_1 \right]_T^S \right] \not\equiv C \left[ \left[ C_2 \right]_T^S \right] \Rightarrow P_1 \not\equiv_{ctx} P_2 \]

unfold \( \sim_{ctx} \)

contrapositive
we have:

- \( C_1 \sim_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \)

we need to prove

- \( \exists C. \ C \left[ [C_1]^S_T \right] \not\vdash C \left[ [C_2]^S_T \right] \Rightarrow \exists C. C[C_2] \not\vdash C[C_2] \)

unfold \( \sim_{ctx} \)

contrapositive

unfold \( \sim_{ctx} \)
we have:

• $C_1 \approx_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2)$

we need to prove

• $\exists C. C \left[ \llbracket C_1 \rrbracket_T^S \right] \not\models C \left[ \llbracket C_2 \rrbracket_T^S \right] \Rightarrow \exists C. C \llbracket C_2 \rrbracket \not\models C \llbracket C_2 \rrbracket$

• unfold $\approx_{ctx}$

• contraposition

• unfold $\approx_{ctx}$

• backtranslation!
• we have:
  • \( C_1 \sim_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \)
• we need to prove
  • \( \exists C. C \left[ [C_1]^S_T \right] \nmid C \left[ [C_2]^S_T \right] \Rightarrow \exists C. C \left[ C_2 \right] \nmid C \left[ C_2 \right] \)
• generate \( C \) based on \( C \)
we have:
- \( C_1 \sim_{ctx} C_2 \iff TR(C_1) = TR(C_2) \)

we need to prove
- \( [P_1]^S_T \not\sim_{ctx} [P_2]^S_T \Rightarrow \exists C. C[C_2] \not\in C[C_2] \)

generate \( C \) based on \( C \)
if complex, apply Traces (folding \( \sim_{ctx} \))
we have:
\[ C_1 \sim_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \]

we need to prove
\[ [P_1]^S_T \neq [P_2]^S_T \Rightarrow \exists C. C[C_2] \not\subseteq C[C_2] \]

generate C based on C

if complex, apply Traces (folding \( \sim_{ctx} \))
we have:

\[ C_1 \sim_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \]

we need to prove

\[ \text{TR}(C_1) \neq \text{TR}(C_2) \Rightarrow \exists C. C[C_2] \not\subseteq C[C_2] \]

generate C based on C

if complex, apply Traces (folding \( \sim_{ctx} \))
• we have:
  • \( C_1 \equiv_{ctx} C_2 \iff \text{TR}(C_1) = \text{TR}(C_2) \)

• we need to prove
  • \( \exists \alpha \in \text{TR}(C_1), \alpha \notin \text{TR}(C_2) \Rightarrow \exists C. C[C_2] \not\vdash C[C_2] \)

• generate \( C \) based on \( C \)

• if complex, apply Traces (folding \( \equiv_{ctx} \))
Backtranslation at work

to the board