

Lecture 2: Proving Full Abstraction (+ Question)

CS350

Marco Patrignani

Some Answers: CEQ with Randomisation

- Assume the language has rand

Some Answers: CEQ with Randomisation

- Assume the language has rand

- $$\frac{\text{(Rand)} \\ n \in \mathbb{N}}{\text{rand} \rightarrow n}$$

Some Answers: CEQ with Randomisation

- Assume the language has rand

$$\bullet \frac{\text{(Rand)} \\ n \in \mathbb{N}}{\text{rand} \rightarrow n}$$

```
1 public Int random(){return rand;} // P1
```

```
1 public Int random(){rand; return rand;} // P2
```

```
1 public Int random(){return < rand;, rand;> } // P3
```

```
1 public Int random(){x=rand; return < x,x> ;} // P4
```

Some Answers: CEQ with Randomisation

- Assume the language has `rand`

$$\bullet \frac{\text{(Rand)} \quad n \in \mathbb{N}}{\text{rand} \rightarrow n}$$

```
1 public Int random(){return rand;} // P1
```

```
1 public Int random(){rand; return rand;} // P2
```

```
1 public Int random(){return < rand;, rand;> } // P3
```

```
1 public Int random(){x=rand; return < x,x> ;} // P4
```

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has `rand`

$$\frac{\text{(Rand)} \quad n \in \mathbb{N}}{\text{rand} \rightarrow n}$$

```
1 public
```

Q: are P_1 and P_2 equivalent?

```
1 public
```

```
1 public Int random(){return < rand;, rand;> } // P3
```

```
1 public Int random(){x=rand; return < x,x> ;} // P4
```

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has `rand`

- $$\frac{\text{(Rand)} \quad n \in \mathbb{N}}{\text{rand} \rightarrow n}$$

```
1 public
```

Q: are P_1 and P_2 equivalent?

Should they be?

```
1 public
```

```
1 public Int random(){return < rand;, rand;> } // P3
```

```
1 public Int random(){x=rand; return < x,x> ;} // P4
```

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has `rand`

- $$\frac{\text{(Rand)} \quad n \in \mathbb{N}}{O; n \triangleright \text{rand} \rightarrow O \triangleright n}$$
- **Oracles:** infinite lists of random numbers

```
1 public Int random(){return rand;} // P1
```

```
1 public Int random(){rand; return rand;} // P2
```

```
1 public Int random(){return < rand;, rand;> } // P3
```

```
1 public Int random(){x=rand; return < x,x> ;} // P4
```

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has rand

CEQ:

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}. \mathfrak{C}[P_1] \downarrow \iff \mathfrak{C}[P_2] \downarrow$$

publi

publi

publi

publi

3

P4

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\simeq_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has rand

CEQ:

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}. \mathfrak{C}[P_1] \downarrow \iff \mathfrak{C}[P_2] \downarrow$$

CEQ-with-rand, try 1:

$$P_1 \simeq_{ctx} ?P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}, \forall O.O \triangleright \mathfrak{C}[P_1] \downarrow \iff \\ O \triangleright \mathfrak{C}[P_2] \downarrow$$

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\simeq_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has rand

CEQ:

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}. \mathfrak{C}[P_1] \downarrow \iff \mathfrak{C}[P_2] \downarrow$$

CEQ-with-rand, try 1:

$$P_1 \simeq_{ctx} ?P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}, \forall O.O \triangleright \mathfrak{C}[P_1] \downarrow \iff \\ O \triangleright \mathfrak{C}[P_2] \downarrow$$

No!

P_1 and P_2 are not equivalent with this definition (but they should be)

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\simeq_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has rand

- Contextual Preorder:
(not an eq, not symmetric)

$$P_1 \sqsubseteq P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}, \forall O_1. O_1 \triangleright \mathfrak{C}[P_1] \downarrow \Rightarrow \exists O_2. O_2 \triangleright \mathfrak{C}[P_2] \downarrow$$

publi

publi

publi

publi

3

P4

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Some Answers: CEQ with Randomisation

Contextual Preorder:

(not an eq, not symmetric)

$$P_1 \sqsubseteq P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}, \forall O_1. O_1 \triangleright \mathfrak{C}[P_1] \downarrow \Rightarrow \\ \exists O_2. O_2 \triangleright \mathfrak{C}[P_2] \downarrow$$

For $P_1 \simeq_{ctx} P_2$, O_2 is O_1 with every other element interleaved with random numbers

publi

publi

publi

publi

Intui

3

P4

Some Answers: CEQ with Randomisation

- Assume the language has rand

Contextual Preorder:

(not an eq, not symmetric)

$$P_1 \sqsubseteq P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}, \forall O_1. O_1 \triangleright \mathfrak{C}[P_1] \downarrow \Rightarrow \\ \exists O_2. O_2 \triangleright \mathfrak{C}[P_2] \downarrow$$

Must also include $P_2 \sqsubseteq P_1$ otherwise P_3 and P_4 are also equivalent (and they should not be)

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Some Answers: CEQ with Randomisation

- Assume the language has rand

Contextual Preorder:

(not an eq, not symmetric)

$$P_1 \sqsubseteq P_2 \stackrel{\text{def}}{=} \forall \mathfrak{C}, \forall O_1. O_1 \triangleright \mathfrak{C}[P_1] \downarrow \Rightarrow \\ \exists O_2. O_2 \triangleright \mathfrak{C}[P_2] \downarrow$$

Must also include $P_2 \sqsubseteq P_1$ otherwise P_3 and P_4 are also equivalent (and they should not be)

$$P_1 \simeq_{ctx} P_2 \stackrel{\text{def}}{=} P_1 \sqsubseteq P_2 \cap P_2 \sqsubseteq P_1$$

Intuitively $P_1 \simeq_{ctx} P_2$ and $P_3 \not\approx_{ctx} P_4$

Other Equivalences

- Contextual equivalence is **not** the only notion of program equivalence

Other Equivalences

- Contextual equivalence is **not** the only notion of program equivalence
- **Any** semantics defines its notion of equivalence

Other Equivalences

- Contextual equivalence is **not** the only notion of program equivalence
- **Any** semantics defines its notion of equivalence
- **Any** notion of equivalence can be used in the statement of fully abstract compilation

Other Equivalences

- Contextual equivalence is **not** the only notion of program equivalence
- **Any** semantics defines its notion of equivalence
- **Any** notion of equivalence can be used in the statement of fully abstract compilation
- Trace semantics or bisimilarity are widely used

Fully Abstract Compilation

$\llbracket \cdot \rrbracket_{\mathbf{T}}^S$ is FAC $\stackrel{\text{def}}{=} \forall P_1, P_2$

$$P_1 \simeq_{ctx} P_2 \iff \llbracket P_1 \rrbracket_{\mathbf{T}}^S \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^S$$

Fully Abstract Compilation

$\llbracket \cdot \rrbracket_{\mathbf{T}}^{\mathbf{S}}$ is FAC $\stackrel{\text{def}}{=} \forall P_1, P_2$

$$P_1 \simeq_{ctx} P_2 \iff \llbracket P_1 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^{\mathbf{S}}$$

- break the \iff :
 1. \Rightarrow : $\forall P_1, P_2. P_1 \simeq_{ctx} P_2 \Rightarrow \llbracket P_1 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^{\mathbf{S}}$
 2. \Leftarrow : $\forall P_1, P_2. \llbracket P_1 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \Rightarrow P_1 \simeq_{ctx} P_2$
- point 2 (should) follow from compiler correctness

Fully Abstract Compilation

$\llbracket \cdot \rrbracket_{\mathbf{T}}^{\mathbf{S}}$ is FAC $\stackrel{\text{def}}{=} \forall P_1, P_2$

$$P_1 \simeq_{ctx} P_2 \iff \llbracket P_1 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^{\mathbf{S}}$$

- break the \iff :
 1. \Rightarrow : $\forall P_1, P_2. P_1 \simeq_{ctx} P_2 \Rightarrow \llbracket P_1 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^{\mathbf{S}}$
 2. \Leftarrow : $\forall P_1, P_2. \llbracket P_1 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^{\mathbf{S}} \Rightarrow P_1 \simeq_{ctx} P_2$
- point 2 (should) follow from compiler correctness
- point 1 is tricky, because of \simeq_{ctx} and its $\forall \mathcal{C}$

Fully Abstract Compilation

$\llbracket \cdot \rrbracket_{\mathbf{T}}^S$ is FAC $\stackrel{\text{def}}{=} \forall P_1, P_2$

$$P_1 \simeq_{ctx} P_2 \iff \llbracket P_1 \rrbracket_{\mathbf{T}}^S \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^S$$

- break the \iff :
 1. \Rightarrow : $\forall P_1, P_2. P_1 \simeq_{ctx} P_2 \Rightarrow \llbracket P_1 \rrbracket_{\mathbf{T}}^S \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^S$
 2. \Leftarrow : $\forall P_1, P_2. \llbracket P_1 \rrbracket_{\mathbf{T}}^S \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^S \Rightarrow P_1 \simeq_{ctx} P_2$
- point 2 (should) follow from compiler correctness
- point 1 is tricky, because of \simeq_{ctx} and its $\forall \mathcal{C}$
This structure is called a **backtranslation**

Trace Semantics

- we replace \simeq_{ctx} with something **equivalent**

Trace Semantics

- we replace \simeq_{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)

Trace Semantics

- 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 \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
- a **trace** semantics

Traces for a program

main method

this is code written by
the attacker

■ function definition
of our code

private data of our program

other code

written by the attacker
(this is the context 🇸🇰!)

- interest in the behaviour of our code (component)

Traces for a program

main method
this is code written by
the attacker

function definition
of our code

private data of our program

other code
written by the attacker
(this is the context 🇸🇰!)

- interest in the behaviour of our code (component)
- need to consider the *rest*

Traces for a program

main method

this is code written by
the attacker

function definition
of our code

private data of our program

other code

written by the attacker
(this is the context 🇸🇰!)

- interest in the behaviour of our code (component)
- need to consider the *rest*

Trace Semantics for Our Program

main method
this is code written by
the attacker

■ function definition
of our code

private data of our program

other code
written by the attacker
(this is the context 🚩!)

- disregard the rest

Trace Semantics for Our Program

```
main method  
  this is code written by  
  the attacker
```

```
function definition  
  of our code
```

```
private data of our program
```

```
other code  
  written by the attacker  
  (this is the context 🚩!)
```


- disregard the rest

Trace Semantics for Our Program

```
main method  
  this is code written by  
  the attacker
```

```
function definition  
  of our code
```

```
private data of our program
```

```
other code  
  written by the attacker  
  (this is the context )
```

- disregard the rest
- abstract its behaviour **from the component perspective:**

Trace Semantics for Our Program



The diagram shows a dark background with light-colored text representing code. A yellow callout box labeled "call args." has an arrow pointing to a light gray box. The light gray box contains the text "function definition of our code" above a horizontal line, and "private data of our program" below it. The background text includes "this is code written by the attacker" and "other code written by the attacker (this is the context 🌀!)"

call args.

function definition
of our code

private data of our program

other code
written by the attacker
(this is the context 🌀!)

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

Trace Semantics for Our Program

call args.

method
this is code written by
the attacker

function definition
of our code

private data of our program

other code
written by the attacker
(this is the context 🌀!)

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

Trace Semantics for Our Program



- disregard the rest
- abstract its behaviour **from the component perspective:**
 1. jump to an entry point ■
- abstract the component behaviour **from the rest perspective:**
 1. call/return

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 $\text{TR}(C) = \left\{ \bar{\alpha} \mid C \xRightarrow{\bar{\alpha}} - \right\}$

Trace Actions

Labels $L ::= a \mid \epsilon$

Observable actions $\alpha ::= \surd \mid g? \mid g!$

Actions $g ::= \text{call } f(v) \mid \text{ret } v$

Traces for a program

We need to define:

- trace states (almost program states)
- labels that make traces
- rules for generating labels and traces ...
- the traces of a component $TR(C) = \dots$

Trace Equivalence

- all semantics yield a notion of **equivalence**

Trace Equivalence

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

$$C_1 \simeq_{ctx} C_2$$

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**

$$C_1 \stackrel{\text{I}}{=} C_2$$

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**

$$TR(C_1) = TR(C_2)$$

the traces of C_1 are the same of those of C_2 11

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**

$$\left\{ \bar{\alpha} \mid C_1 \xRightarrow{\bar{\alpha}} - \right\} = \left\{ \bar{\alpha} \mid C_2 \xRightarrow{\bar{\alpha}} - \right\}$$

the traces of C_1 are the same of those of C_2

Proofs about Trace Semantics

- **any** trace semantics won't just work
- it needs to be **correct** and **complete**

Proofs about Trace Semantics

- any trace semantics won't just work
- it needs to be correct and complete

$$C_1 \simeq_{ctx} C_2 \iff C_1 \stackrel{I}{=} C_2$$

Proofs about Trace Semantics

- **any** trace semantics won't just work
- it needs to be **correct** (\Leftarrow) and **complete** (\Rightarrow)

$$C_1 \simeq_{ctx} C_2 \iff C_1 \stackrel{I}{=} C_2$$

Fully Abstract Compilation & Target Traces

- we have:

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

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $P_1 \simeq_{ctx} P_2 \Rightarrow \llbracket P_1 \rrbracket_{\mathbf{T}}^S \simeq_{ctx} \llbracket P_2 \rrbracket_{\mathbf{T}}^S$

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $P_1 \simeq_{ctx} P_2 \Rightarrow \forall e. e \llbracket [C_1]_T^S \rrbracket \downarrow \iff e \llbracket [C_2]_T^S \rrbracket \downarrow$
- unfold \simeq_{ctx}

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\exists e. e[\llbracket C_1 \rrbracket_T^S] \downarrow \not\Rightarrow e[\llbracket C_2 \rrbracket_T^S] \downarrow \Rightarrow P_1 \not\sim_{ctx} P_2$
- unfold \simeq_{ctx}
- contrapositive

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\exists e. e[\llbracket C_1 \rrbracket_T^S] \downarrow \not\iff e[\llbracket C_2 \rrbracket_T^S] \downarrow \Rightarrow$
 $\exists e. e[C_2] \downarrow \not\iff e[C_2] \downarrow$
- $unfold \simeq_{ctx}$
- contrapositive
- $unfold \simeq_{ctx}$

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\exists e. e[\llbracket C_1 \rrbracket_T^S] \downarrow \not\iff e[\llbracket C_2 \rrbracket_T^S] \downarrow \Rightarrow$
 $\exists e. e[C_2] \downarrow \not\iff e[C_2] \downarrow$
- unfold \simeq_{ctx}
- contrapositive
- unfold \simeq_{ctx}
- backtranslation!

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\exists \mathcal{E}. \mathcal{E}[\llbracket C_1 \rrbracket_T^S] \downarrow \not\iff \mathcal{E}[\llbracket C_2 \rrbracket_T^S] \downarrow \Rightarrow$
 $\exists \mathcal{E}. \mathcal{E}[C_2] \downarrow \not\iff \mathcal{E}[C_2] \downarrow$
- generate \mathcal{E} based on \mathcal{E}

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\llbracket P_1 \rrbracket_T^S \not\stackrel{ctx}{\sim} \llbracket P_2 \rrbracket_T^S \Rightarrow \exists \mathcal{C}. \mathcal{C}[C_2] \downarrow \not\iff \mathcal{C}[C_2] \downarrow$
- generate \mathcal{C} based on \mathcal{C}
- if complex, apply Traces (folding \simeq_{ctx})

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\llbracket P_1 \rrbracket_T^S \not\equiv \llbracket P_2 \rrbracket_T^S \Rightarrow \exists \mathcal{C}. \mathcal{C}[C_2] \downarrow \not\equiv \mathcal{C}[C_2] \downarrow$
- generate \mathcal{C} based on \mathcal{C}
- if complex, apply Traces (folding \simeq_{ctx})

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $TR(C_1) \neq TR(C_2) \Rightarrow \exists \mathcal{E}. \mathcal{E}[C_2] \downarrow \not\iff \mathcal{E}[C_2] \downarrow$
- generate \mathcal{E} based on \mathcal{C}
- if complex, apply Traces (folding \simeq_{ctx})

Fully Abstract Compilation & Target Traces

- we have:
 - $C_1 \simeq_{ctx} C_2 \iff TR(C_1) = TR(C_2)$
- we need to prove
 - $\exists \alpha \in TR(C_1), \alpha \notin TR(C_2) \Rightarrow$
 $\exists \mathcal{E}. \mathcal{E}[C_2] \downarrow \not\Leftarrow \mathcal{E}[C_2] \downarrow$
- generate \mathcal{E} based on \mathcal{E}
- if complex, apply Traces (folding \simeq_{ctx})