Lecture 6: Proofs

Secure Compilation Seminar

Marco Patrignani

CISPA
HELMHOLTZ-ZENTRUM i. G.

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?

$$P \Rightarrow Q$$

- 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

$$P \Rightarrow Q$$

- 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

$$P \to Q$$

- 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?

$$\llbracket \cdot \rrbracket^{\mathsf{S}}_{\mathbf{T}} \text{ is FAC} \overset{\text{def}}{=} \forall \mathsf{P}_1, \mathsf{P}_2$$

$$\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \iff \llbracket \mathsf{P}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} \llbracket \mathsf{P}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}$$

Fully Abstract Compilation

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

Fully Abstract Compilation

- break the $\iff$:
    1. $\Rightarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$
    2. $\Leftarrow$: $\forall \mathsf{P}_1, \mathsf{P}_2.\ [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2$
- point 2 (should) follow from compiler correctness
- point 1 is tricky, because of $\simeq_{ctx}$ and its $\forall \mathfrak{C}$

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)

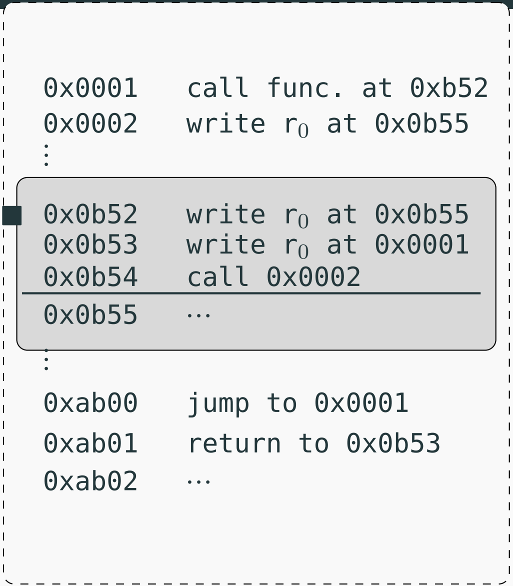
Trace Semantics

- 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 PMA



0x0001 call func. at 0xb52

0x0002 write r_0 at 0x0b55

⋮

0x0b52 write r_0 at 0x0b55

0x0b53 write r_0 at 0x0001

0x0b54 call 0x0002

0x0b55 ⋯

⋮

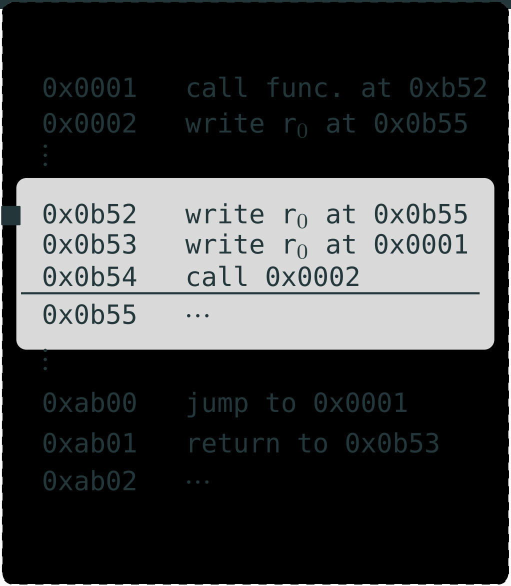
0xab00 jump to 0x0001

0xab01 return to 0x0b53

0xab02 ⋯

- interest in the behaviour of the module

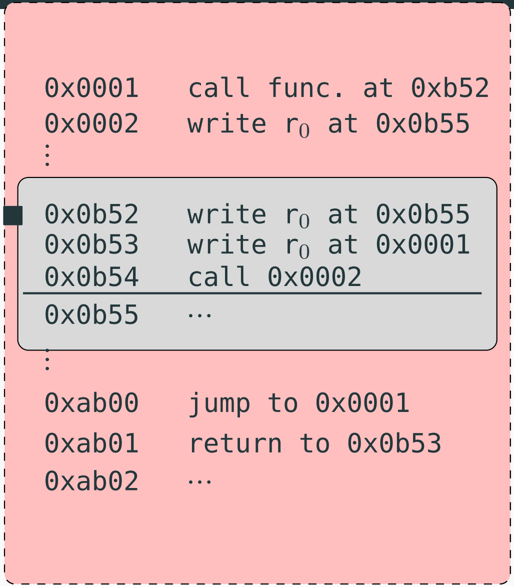
Traces for PMA



0x0001	call func. at 0xb52
0x0002	write r_0 at 0x0b55
⋮	
0x0b52	write r_0 at 0x0b55
0x0b53	write r_0 at 0x0001
0x0b54	call 0x0002
0x0b55	…
⋮	
0xab00	jump to 0x0001
0xab01	return to 0x0b53
0xab02	…

- interest in the behaviour of the module
- need to consider the *rest*

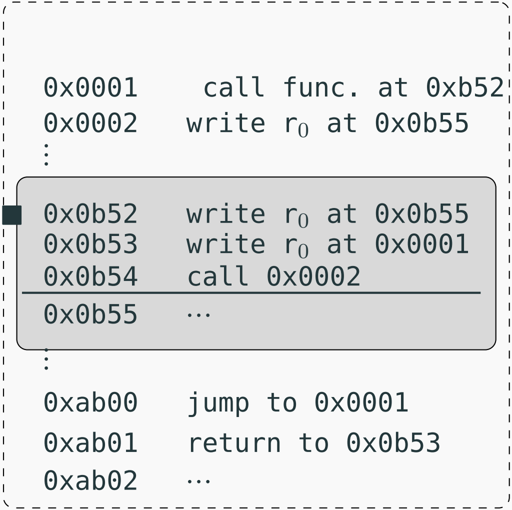
Traces for PMA



0x0001 call func. at 0xb52
0x0002 write r_0 at 0x0b55
⋮
0x0b52 write r_0 at 0x0b55
0x0b53 write r_0 at 0x0001
0x0b54 call 0x0002
0x0b55 …
⋮
0xab00 jump to 0x0001
0xab01 return to 0x0b53
0xab02 …

- interest in the behaviour of the module
- need to consider the *rest*

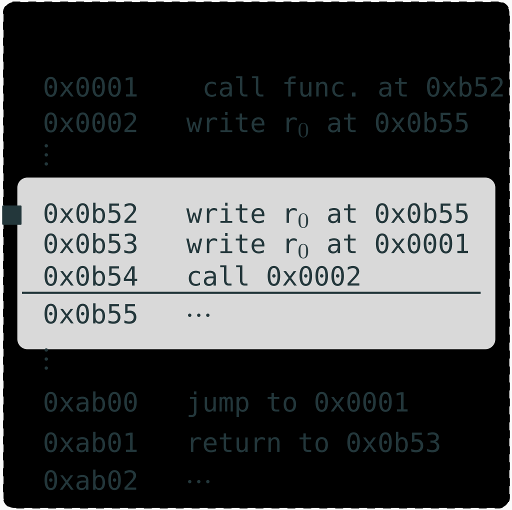
Trace Semantics for PMA



0x0001	call func. at 0xb52
0x0002	write r_0 at 0x0b55
⋮	
0x0b52	write r_0 at 0x0b55
0x0b53	write r_0 at 0x0001
0x0b54	call 0x0002
0x0b55	…
⋮	
0xab00	jump to 0x0001
0xab01	return to 0x0b53
0xab02	…

- disregard the rest

Trace Semantics for PMA



0x0001 call func. at 0xb52

0x0002 write r_0 at 0x0b55

⋮

0x0b52 write r_0 at 0x0b55

0x0b53 write r_0 at 0x0001

0x0b54 call 0x0002

0x0b55 …

⋮

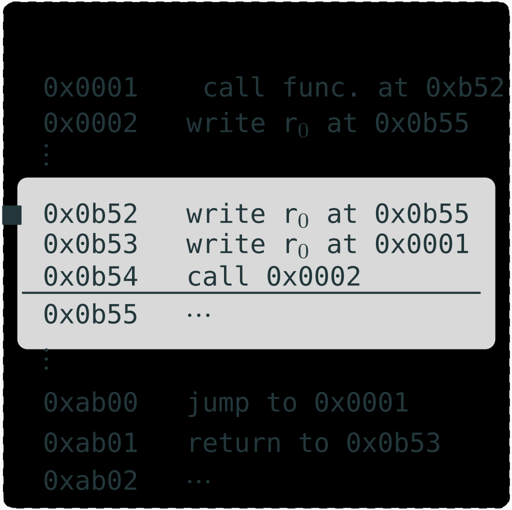
0xab00 jump to 0x0001

0xab01 return to 0x0b53

0xab02 …

- disregard the rest

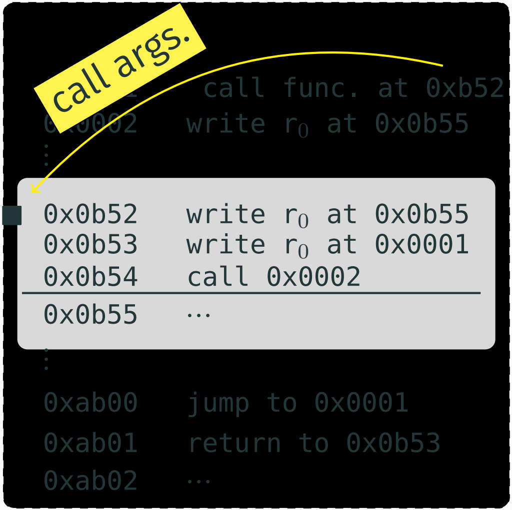
Trace Semantics for PMA



```
0x0001    call func. at 0xb52
0x0002    write r_0 at 0x0b55
⋮
0x0b52    write r_0 at 0x0b55
0x0b53    write r_0 at 0x0001
0x0b54    call 0x0002
0x0b55    …
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    …
```

- disregard the rest
- abstract its behaviour from the module perspective:

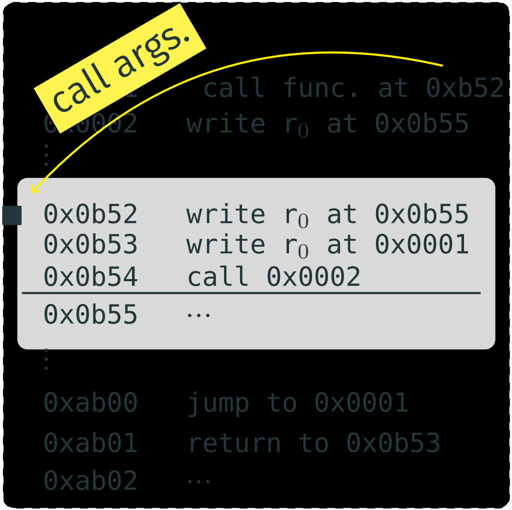
Trace Semantics for PMA



```
           call func. at 0xb52
0x0002     write r0 at 0x0b55
⋮
0x0b52     write r0 at 0x0b55
0x0b53     write r0 at 0x0001
0x0b54     call 0x0002
0x0b55     …
⋮
0xab00     jump to 0x0001
0xab01     return to 0x0b53
0xab02     …
```

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

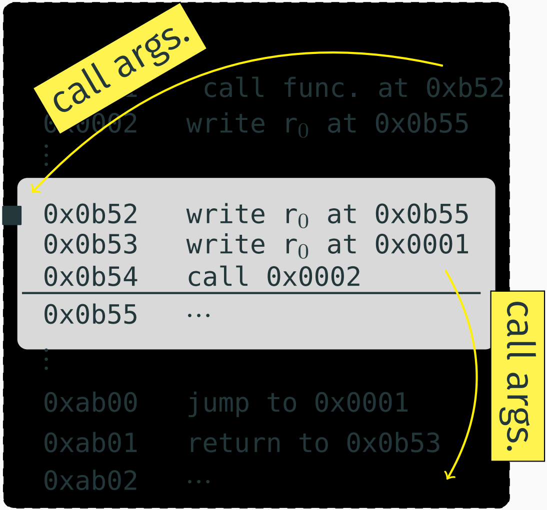
Trace Semantics for PMA



```
       1   call func. at 0xb52
0x0002   write r0 at 0x0b55
⋮
■ 0x0b52   write r0 at 0x0b55
0x0b53   write r0 at 0x0001
0x0b54   call 0x0002
0x0b55   …
⋮
0xab00   jump to 0x0001
0xab01   return to 0x0b53
0xab02   …
```

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

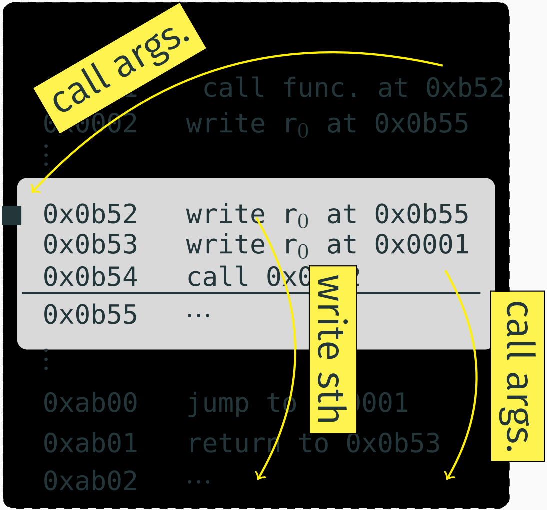
Trace Semantics for PMA



```
           call func. at 0xb52
0x0002     write r0 at 0x0b55
⋮
0x0b52     write r0 at 0x0b55
0x0b53     write r0 at 0x0001
0x0b54     call 0x0002
0x0b55     …
⋮
0xab00     jump to 0x0001
0xab01     return to 0x0b53
0xab02     …
```

- 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



0x0001 call func. at 0xb52
0x0002 write r_0 at 0x0b55
⋮
0x0b52 write r_0 at 0x0b55
0x0b53 write r_0 at 0x0001
0x0b54 call 0x0002
0x0b55 …
⋮
0xab00 jump to 0x0001
0xab01 return to 0x0b53
0xab02 …

- 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 $\mathsf{TR}(C) = \left\{ \overline{\alpha} \;\middle|\; C \overset{\overline{\alpha}}{\Longrightarrow} \_ \right\}$

Trace Actions

$$\begin{aligned}
\mathit{Labels} \quad & L ::= a \mid \epsilon \\
\mathit{Observable\ actions} \quad & \alpha ::= \surd \mid g? \mid g! \\
\mathit{Actions} \quad & g ::= \texttt{call}\ p\ (r) \mid \texttt{ret}\ p\ r(\texttt{r}_0)
\end{aligned}$$

Traces for PMA

We need to define:

- trace states (almost program states) Θ
- labels that make traces
- rules for generating labels and traces $\cdots$
- the traces of a component $\mathsf{TR}(C) = \cdots$

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

$$C_1 \overset{\mathrm{T}}{=} T_2$$

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

$$\mathsf{TR}(C_1) = \mathsf{TR}(C_2)$$

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

$$\left\{ \overline{\alpha} \;\middle|\; C_1 \xRightarrow{\overline{\alpha}} \_ \right\} = \left\{ \overline{\alpha} \;\middle|\; C_2 \xRightarrow{\overline{\alpha}} \_ \right\}$$

the traces of C_1 are the same of those of C_2

Proofs about Trace Semantics

- any trace semantics won’t just work
- 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

$$C_1 \simeq_{ctx} C_2 \iff C_1 \overset{T}{=} C_2$$

Proofs about Trace Semantics

- any trace semantics won't just work
- they need to be
  correct (⇐) and complete (⇒)

$$C_1 \simeq_{ctx} C_2 \iff C_1 \overset{\mathrm{T}}{=} C_2$$

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow [\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}}$

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $\mathsf{P}_1 \simeq_{ctx} \mathsf{P}_2 \Rightarrow \forall \mathbf{C}.\ \mathbf{C}\left[\llbracket \mathsf{C}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \downarrow \mathbf{C}\left[\llbracket \mathsf{C}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right]$
- unfold $\simeq_{ctx}$

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists \mathbf{C}.\ \mathbf{C}\left[[\![\mathsf{C}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \right] \not\downarrow \mathbf{C}\left[[\![\mathsf{C}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \right] \Rightarrow \mathsf{P}_1 \not\simeq_{ctx} \mathsf{P}_2$
- unfold $\simeq_{ctx}$
- contrapositive

Fully Abstract Compilation & Target Traces

- we have:
  - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
  - $\exists\mathbf{C}.\ \mathbf{C}\left[\llbracket \mathsf{C}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \not\Downarrow \mathbf{C}\left[\llbracket \mathsf{C}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \Rightarrow \exists\mathsf{C}.\mathsf{C}\left[\mathsf{C}_2\right] \not\Downarrow \mathsf{C}\left[\mathsf{C}_2\right]$
- unfold $\simeq_{ctx}$
- contrapositive
- unfold $\simeq_{ctx}$

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $\exists \mathbf{C}.\ \mathbf{C}\left[\llbracket \mathsf{C_1} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \not\Downarrow \mathbf{C}\left[\llbracket \mathsf{C_2} \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \Rightarrow \exists \mathsf{C}.\mathsf{C}\left[\mathsf{C_2}\right] \not\Downarrow \mathsf{C}\left[\mathsf{C_2}\right]$
- unfold $\simeq_{ctx}$
- contrapositive
- unfold $\simeq_{ctx}$
- backtranslation!

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $\exists\mathbf{C}.\ \mathbf{C}\left[\llbracket \mathsf{C}_1 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \not\downarrow \mathbf{C}\left[\llbracket \mathsf{C}_2 \rrbracket^{\mathsf{S}}_{\mathbf{T}}\right] \Rightarrow \exists\mathsf{C}.\mathsf{C}\left[\mathsf{C}_2\right] \not\downarrow \mathsf{C}\left[\mathsf{C}_2\right]$
- generate C based on $\mathbf{C}$

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $[\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \not\simeq_{ctx} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \exists \mathsf{C}.\mathsf{C}\left[\mathsf{C}_2\right] \not\downarrow \mathsf{C}\left[\mathsf{C}_2\right]$
- generate C based on $\mathbf{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $[\![\mathsf{P}_1]\!]^{\mathsf{S}}_{\mathbf{T}} \overset{\not\mathsf{T}}{\neq} [\![\mathsf{P}_2]\!]^{\mathsf{S}}_{\mathbf{T}} \Rightarrow \exists \mathsf{C}.\mathsf{C}\left[\mathsf{C}_2\right] \not\downarrow \mathsf{C}\left[\mathsf{C}_2\right]$
- generate C based on $\mathbf{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $\mathsf{TR}(\mathbf{C_1}) \neq \mathsf{TR}(\mathbf{C_2}) \Rightarrow \exists \mathsf{C}.\mathsf{C}[\mathsf{C}_2] \not\downarrow \mathsf{C}[\mathsf{C}_2]$
- generate C based on $\mathbf{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Fully Abstract Compilation & Target Traces

- we have:
    - $\mathbf{C_1} \simeq_{ctx} \mathbf{C_2} \iff \mathsf{TR}(\mathbf{C_1}) = \mathsf{TR}(\mathbf{C_2})$
- we need to prove
    - $\exists \alpha \in \mathsf{TR}(\mathbf{C_1}), \alpha \notin \mathsf{TR}(\mathbf{C_2}) \Rightarrow \exists \mathsf{C}.\mathsf{C}[\mathsf{C}_2] \not\simeq \mathsf{C}[\mathsf{C}_2]$
- generate C based on $\mathbf{C}$
- if complex, apply Traces (folding $\simeq_{ctx}$)

Backtranslation at work

to the board