On Modular and Fully-Abstract Compilation

Marco Patrignani[1] Dominique Devriese[2] Frank Piessens[2]

[1]MPI-SWS, Saarbrücken, Germany
[2]iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium
first.last@[mpi-sws.org | cs.kuleuven.be]

June 2016

Goals

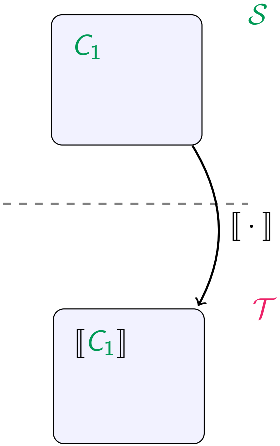
1. understand secure compilation failures due to linking

Goals

1. understand secure compilation failures due to linking
2. present solutions to them relying on Hardware isolation

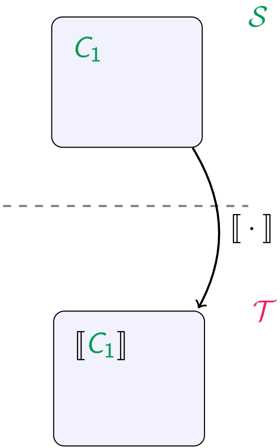
What is a Secure Compiler?

- *compiler*: function from source to target **components**



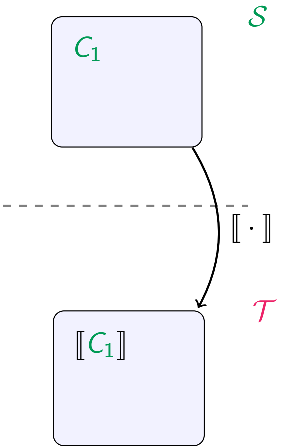
What is a Secure Compiler?

- *compiler*: function from source to target **components**
- *secure compiler*: preserves source-level security properties in the generated components



What is a Secure Compiler?

- *compiler*: function from source to target **components**
- *secure compiler*: preserves source-level security properties in the generated components
- *literature example*: fully-abstract compiler



Fully Abstract Compilation

Fully abstract compilers preserve (and reflect) source-level behaviour in compiled components

Fully Abstract Compilation

Fully abstract compilers preserve (and reflect) source-level behaviour in compiled components

- program behaviour captures security properties e.g., confidentiality, integrity, etc

Fully Abstract Compilation

Fully abstract compilers preserve (and reflect) source-level behaviour in compiled components

- program behaviour captures security properties e.g., confidentiality, integrity, etc
- behaviour preservation (and reflection) means preservation of security properties

Indicating behaviour: Contextual equivalence

- behavioural equivalence = contextual equivalence ($\simeq^{\mathcal{L}}$)

Indicating behaviour: Contextual equivalence

- behavioural equivalence = contextual equivalence ($\simeq^{\mathcal{L}}$)
- $C_1 \simeq^{\mathcal{L}} C_2 \triangleq \forall \mathbb{C},\ \mathbb{C}[C_1] \Uparrow \iff \mathbb{C}[C_2] \Uparrow$

Fully Abstract Compilation Formally

$\forall C_1, C_2 \in \mathcal{S}.$

$$C_1 \simeq^{\mathcal{S}} C_2$$

$$\Updownarrow$$

Fully Abstract Compilation Formally

$\forall C_1, C_2 \in \mathcal{S}.$

$$C_1 \simeq^{\mathcal{S}} C_2$$

$$\Updownarrow$$

$$[\![C_1]\!] \simeq^{\mathcal{T}} [\![C_2]\!]$$

Fully Abstract Compilation Formally

$$\forall C_1, C_2 \in \mathcal{S}.$$

$$(\forall \mathbb{C}.\ \mathbb{C}[C_1] \Uparrow \iff \mathbb{C}[C_2] \Uparrow)$$

$$\Updownarrow$$

$$(\forall \mathbb{C}.\ \mathbb{C}[[\![C_1]\!]] \Uparrow \iff \mathbb{C}[[\![C_2]\!]] \Uparrow)$$

Fully Abstract Compilation Formally

$$\forall C_1, C_2 \in \mathcal{S}.$$

$$(\forall \mathbb{C}.\ \mathbb{C}[C_1] \Uparrow \iff \mathbb{C}[C_2] \Uparrow)$$

$$\Updownarrow$$

$$(\forall \mathbb{C}.\ \mathbb{C}[[\![C_1]\!]] \Uparrow \iff \mathbb{C}[[\![C_2]\!]] \Uparrow)$$

- $\mathbb{C}$ models the attacker

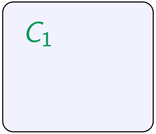
Fully Abstract Compilation Formally

$$\forall C_1, C_2 \in \mathcal{S}.$$

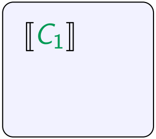
$$(\forall \mathbb{C}.\ \mathbb{C}[C_1]\Uparrow \iff \mathbb{C}[C_2]\Uparrow)$$
$$\Updownarrow$$
$$(\forall \mathbb{C}.\ \mathbb{C}[[\![C_1]\!]]\Uparrow \iff \mathbb{C}[[\![C_2]\!]]\Uparrow)$$

- $\mathbb{C}$ models the attacker
- *attacker model*: protection against code injection attacks

Compiler INsecurity

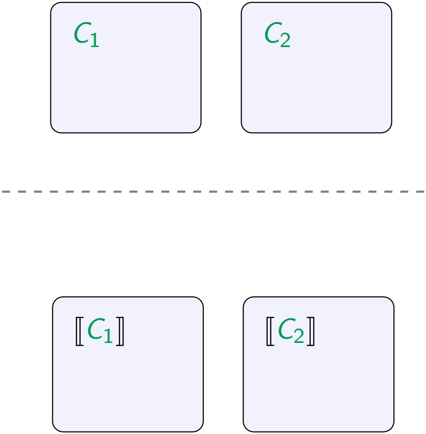


- compiler full-abstraction is often studied in simple settings



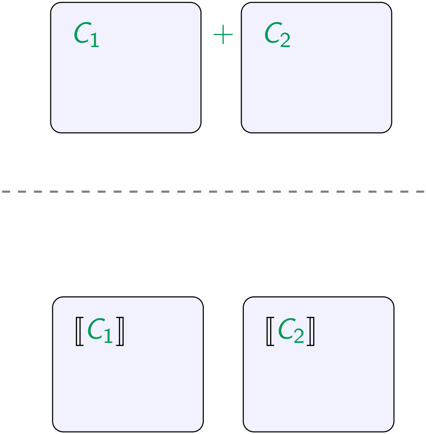
Compiler INsecurity

- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular



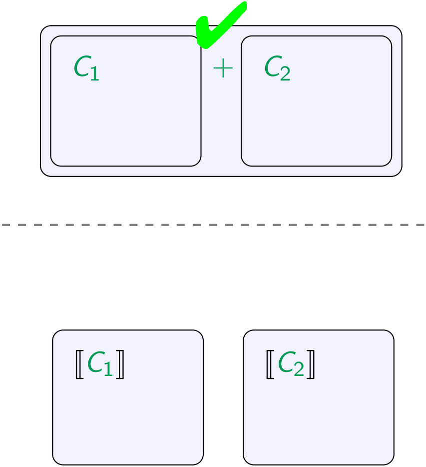
Compiler INsecurity

- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular

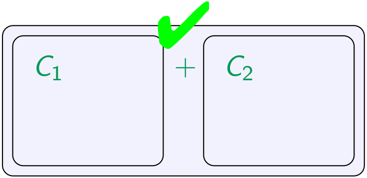


Compiler INsecurity

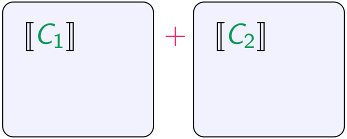
- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular



Compiler INsecurity

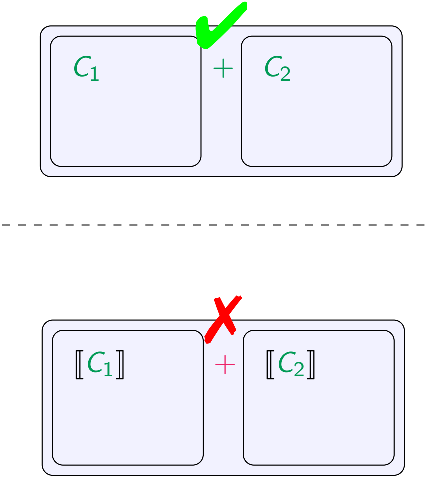


- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular



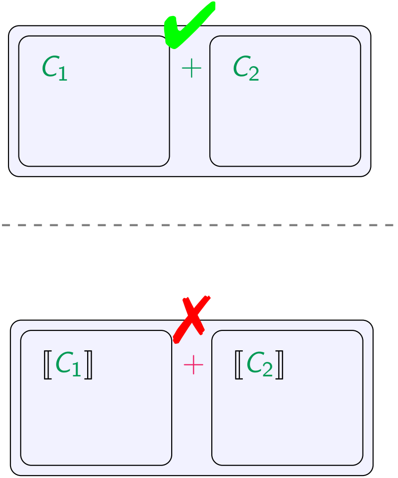
Compiler INsecurity

- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular



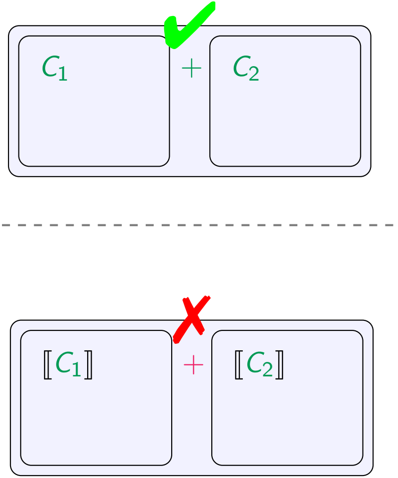
Compiler INsecurity

- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular
- we consider components that *trust* each other and have *shared invariants*

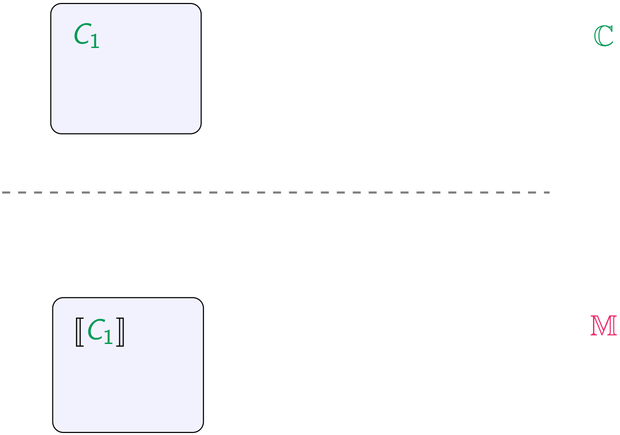


Compiler INsecurity

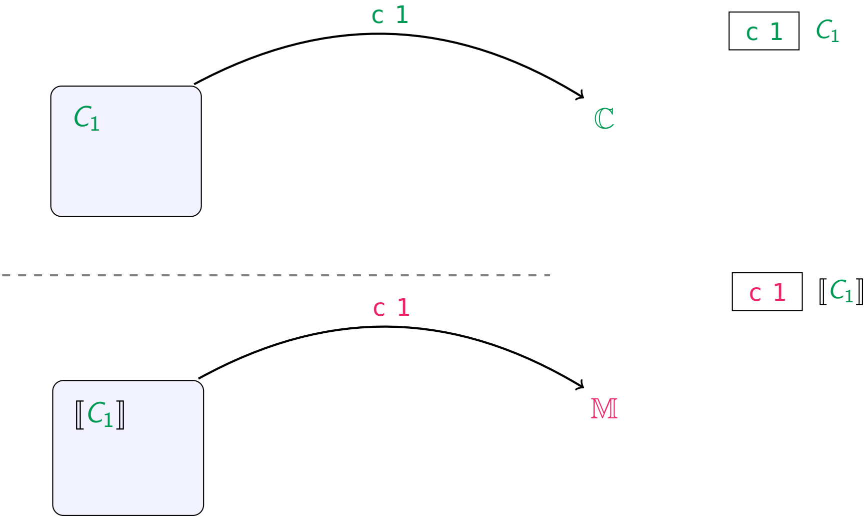
- compiler full-abstraction is often studied in simple settings
- many fully-abstract compilers are not modular
- we consider components that *trust* each other and have *shared invariants*
- we adopt an *object − based language* and an *assembly* one



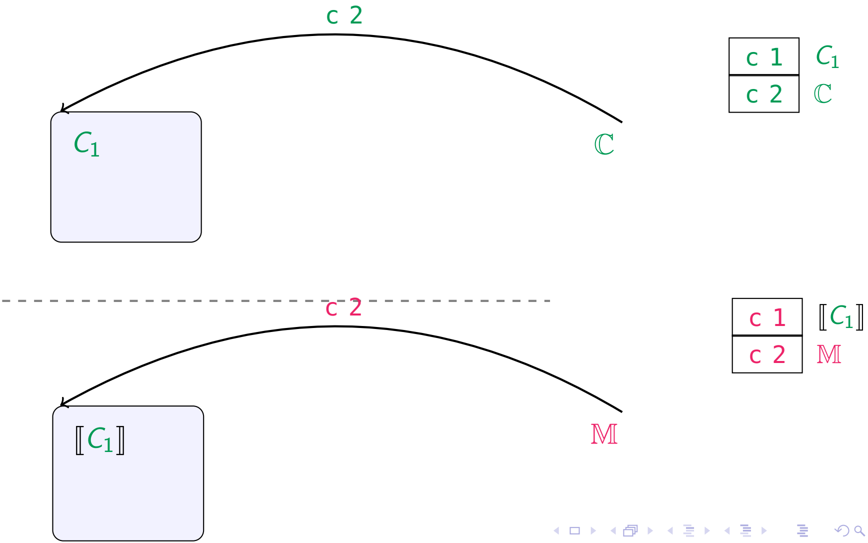
Call Stack Shortcutting - 1 module



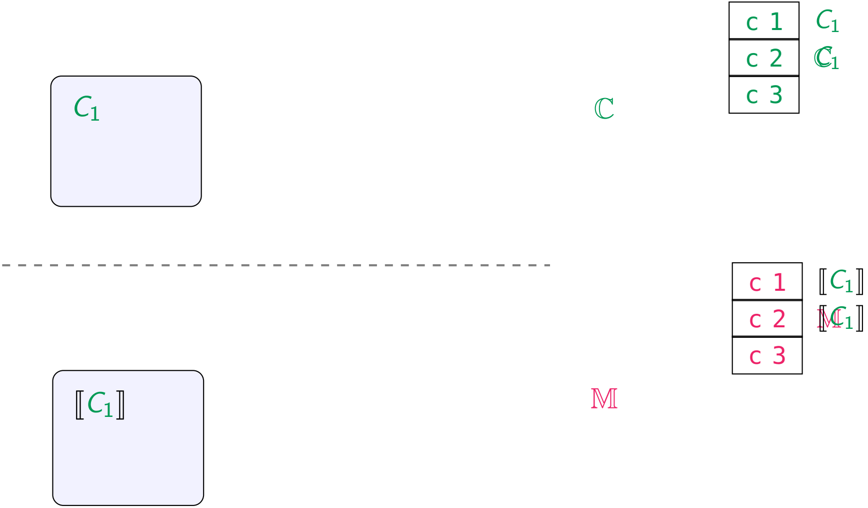
Call Stack Shortcutting - 1 module



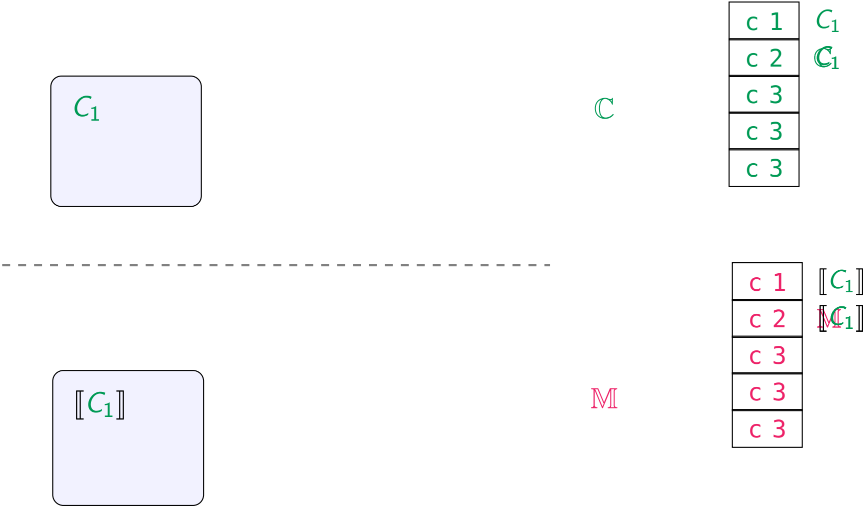
Call Stack Shortcutting - 1 module



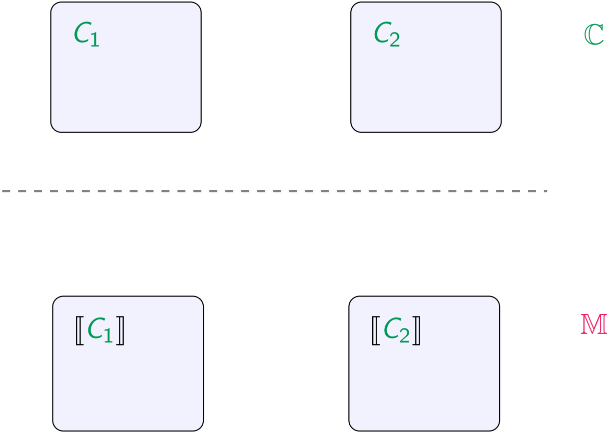
Call Stack Shortcutting - 1 module



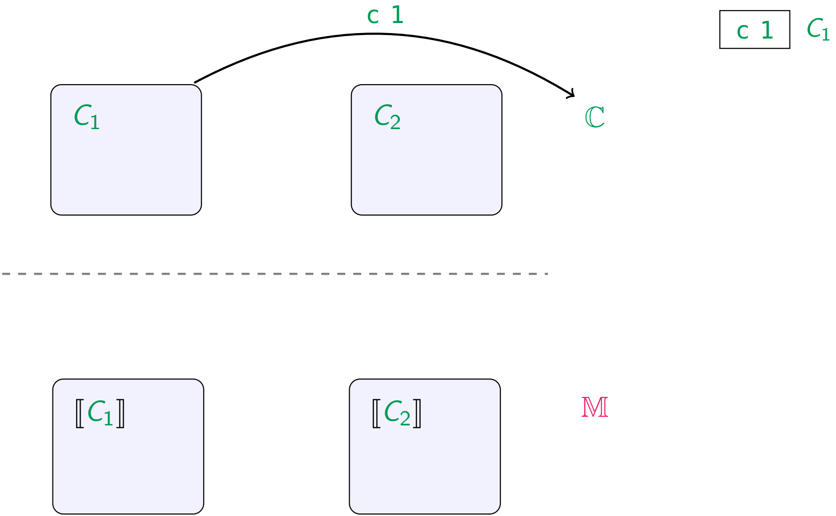
Call Stack Shortcutting - 1 module



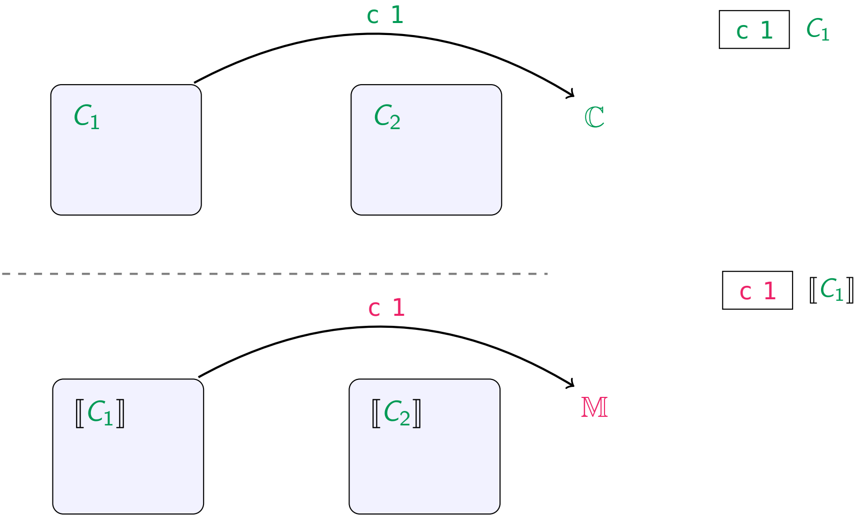
Call Stack Shortcutting



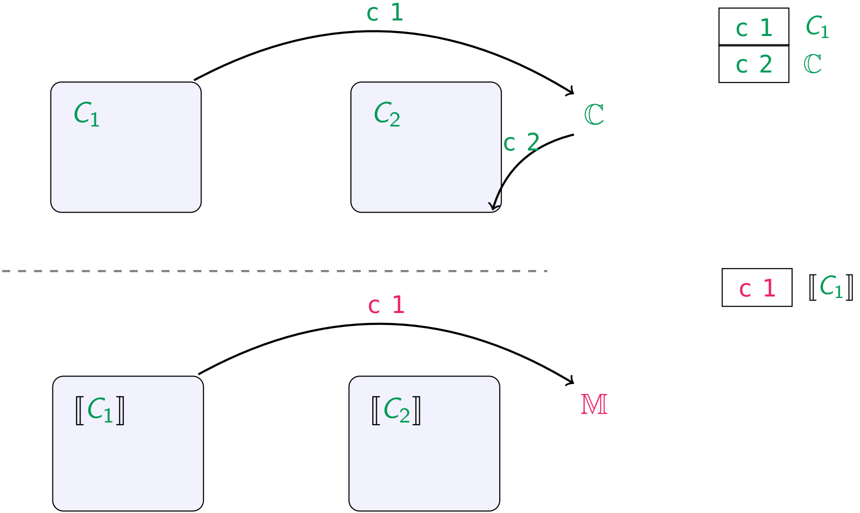
Call Stack Shortcutting



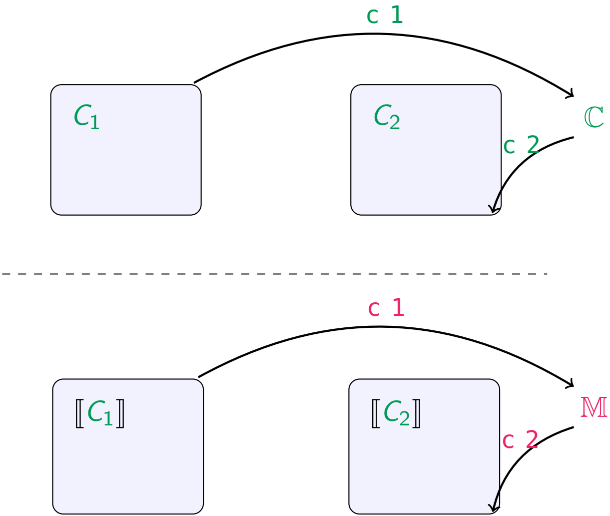
Call Stack Shortcutting



Call Stack Shortcutting



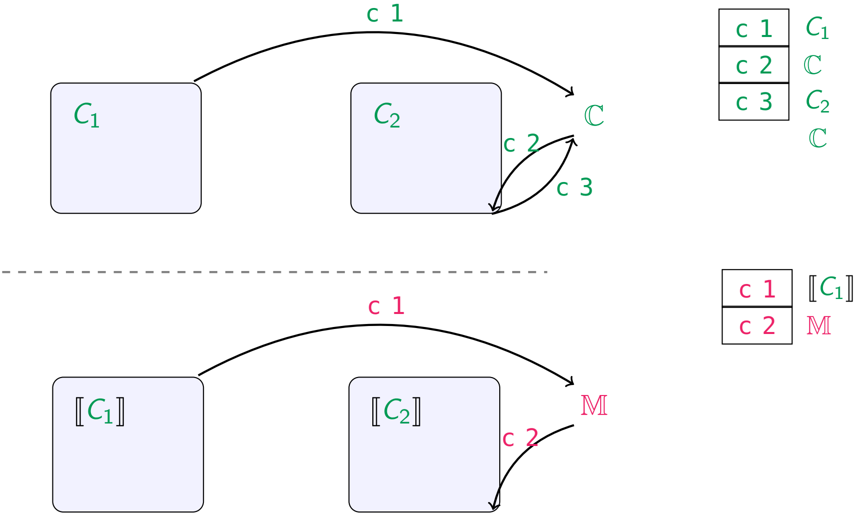
Call Stack Shortcutting



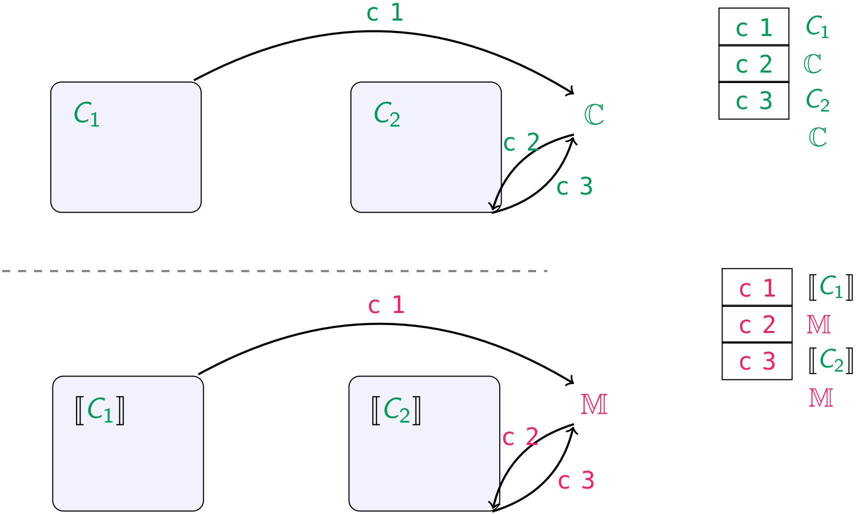
c 1	C_1
c 2	$\mathbb{C}$

c 1	$[\![C_1]\!]$
c 2	$\mathbb{M}$

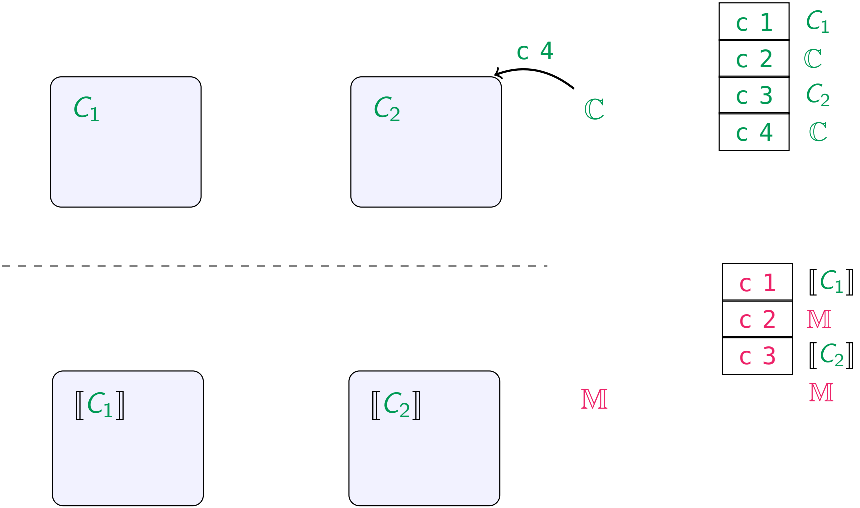
Call Stack Shortcutting



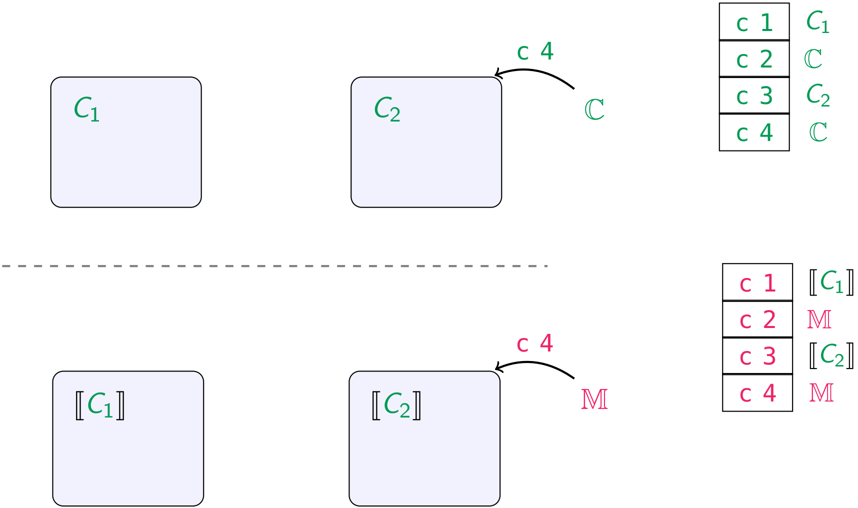
Call Stack Shortcutting



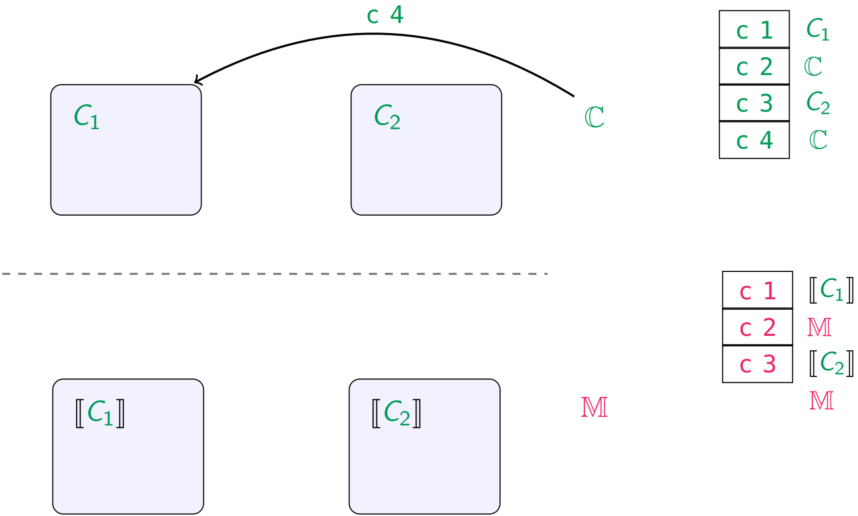
Call Stack Shortcutting



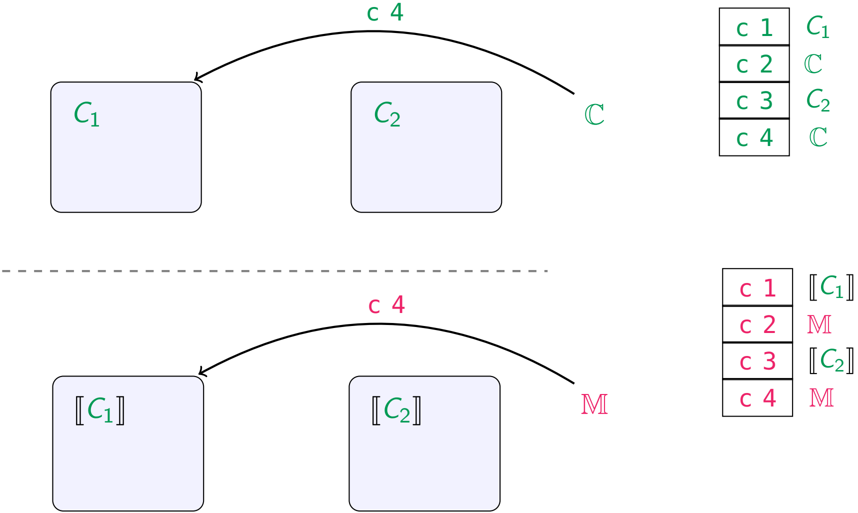
Call Stack Shortcutting



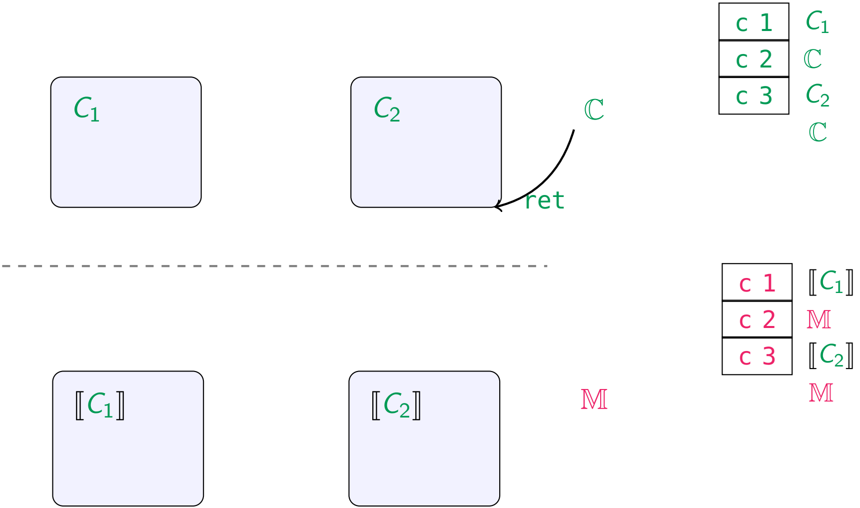
Call Stack Shortcutting



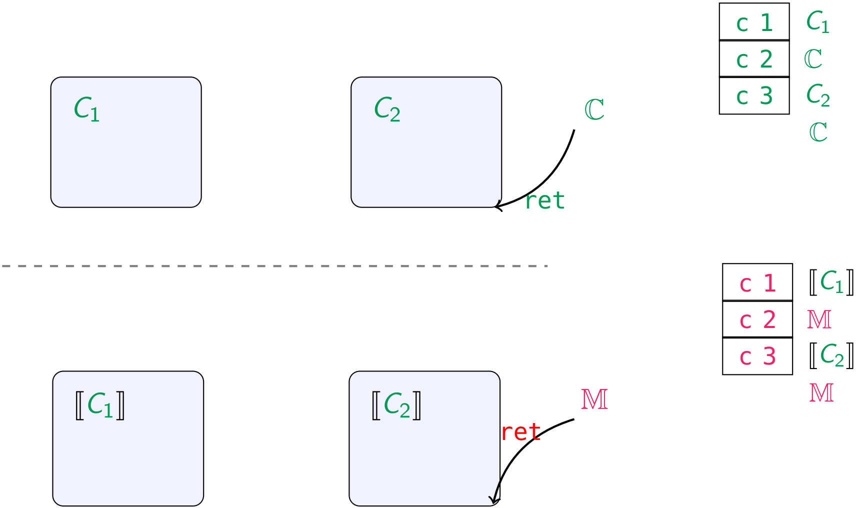
Call Stack Shortcutting



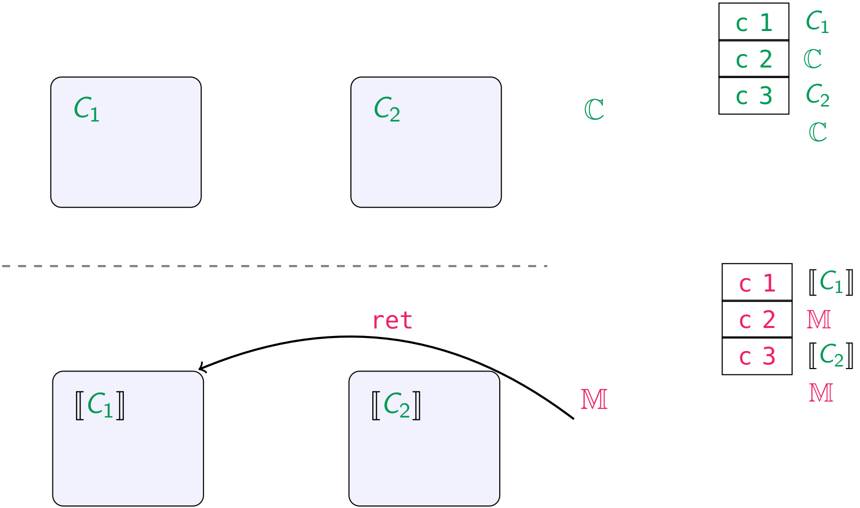
Call Stack Shortcutting



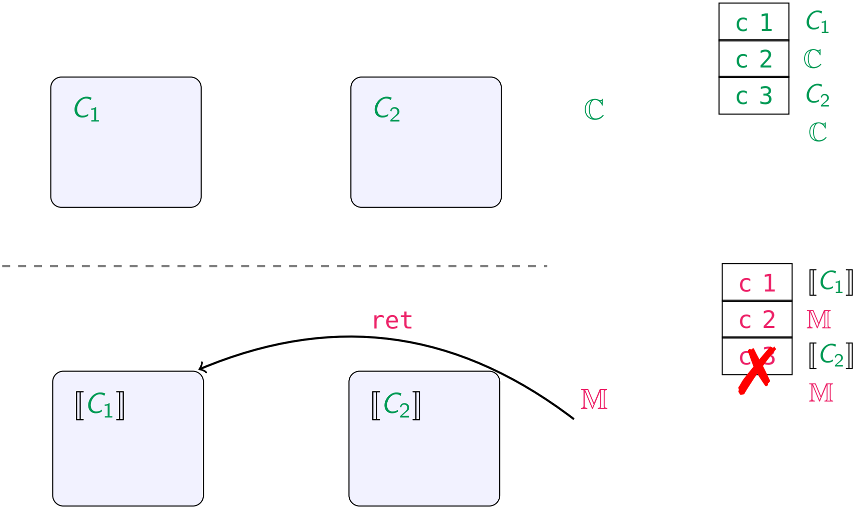
Call Stack Shortcutting



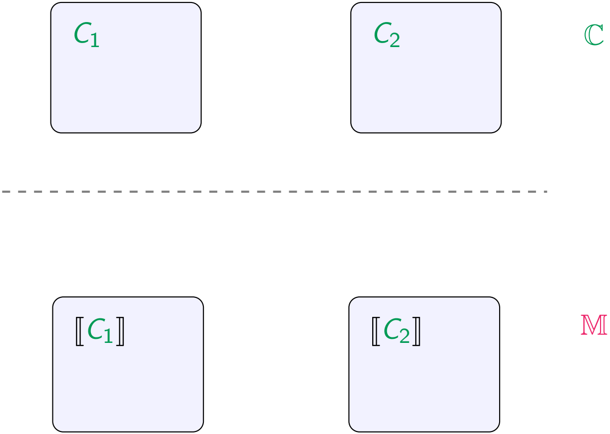
Call Stack Shortcutting



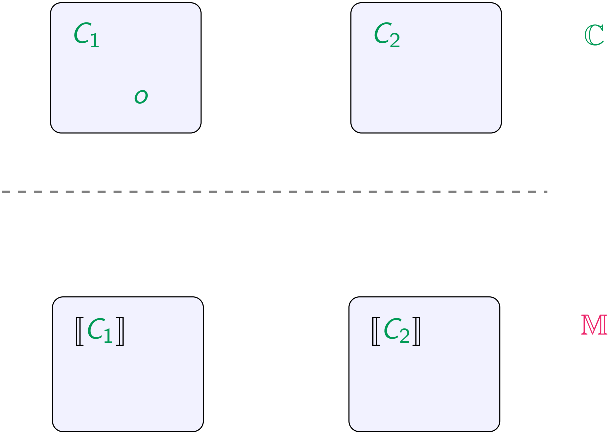
Call Stack Shortcutting



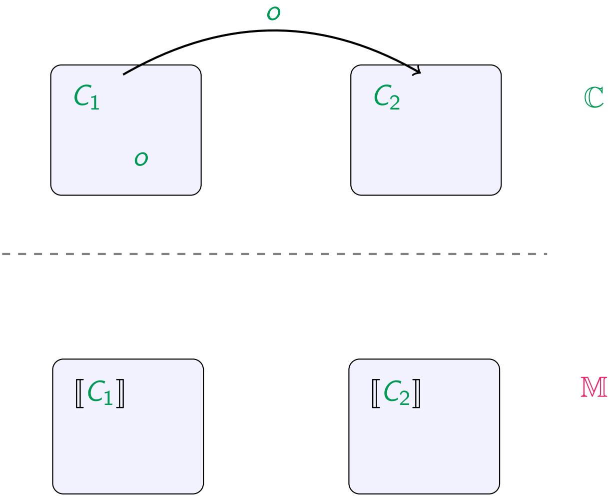
Object Guessing



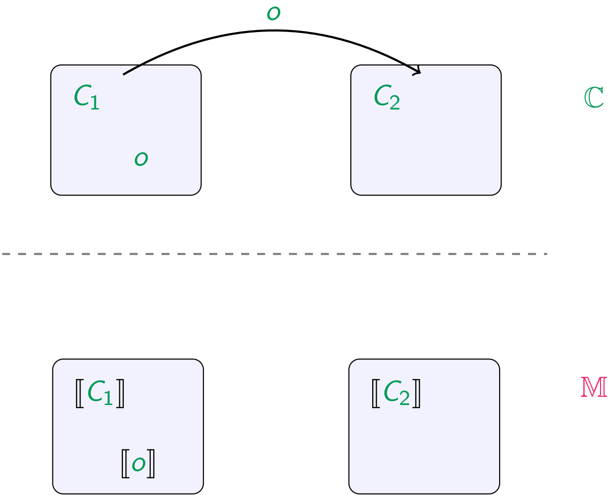
Object Guessing



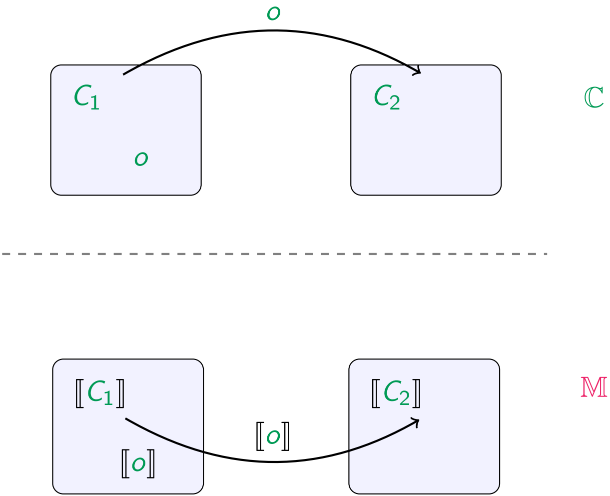
Object Guessing



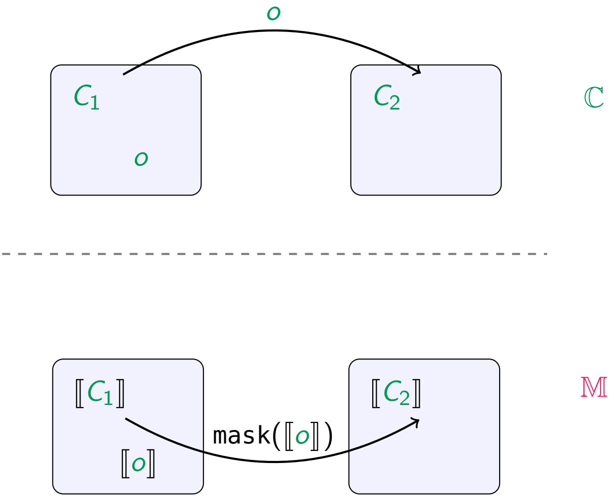
Object Guessing



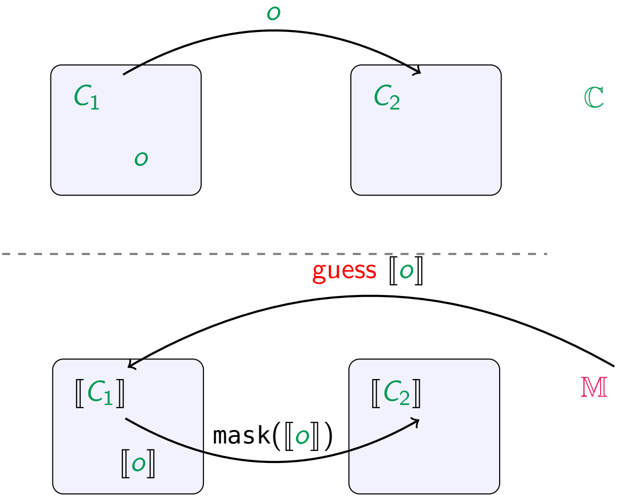
Object Guessing



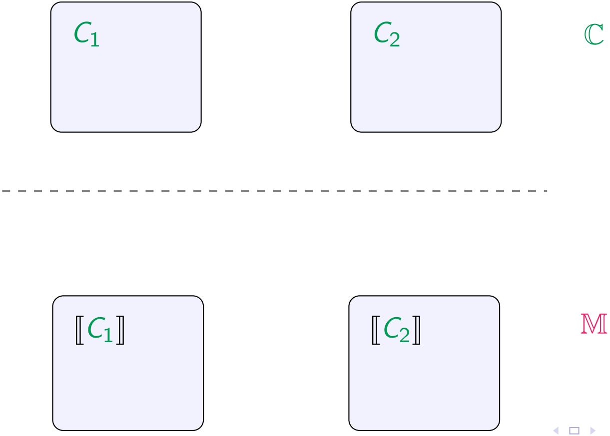
Object Guessing



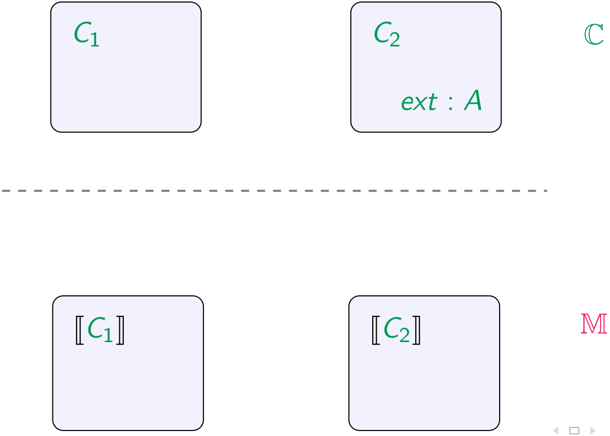
Object Guessing



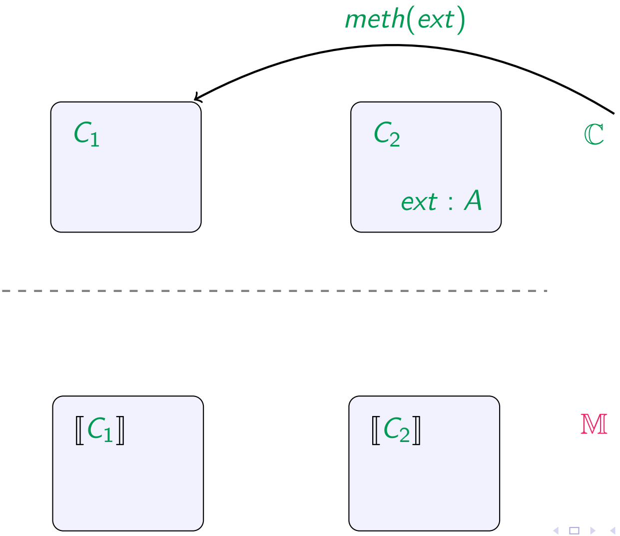
Object Faking



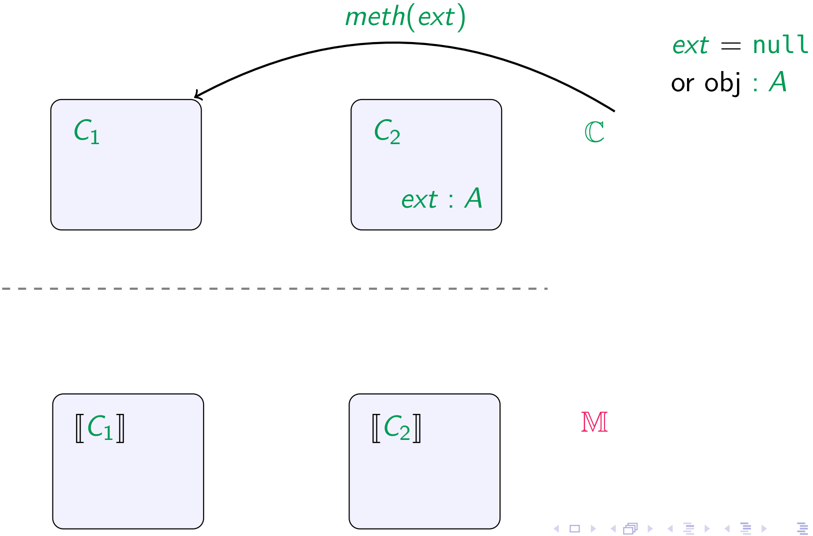
Object Faking



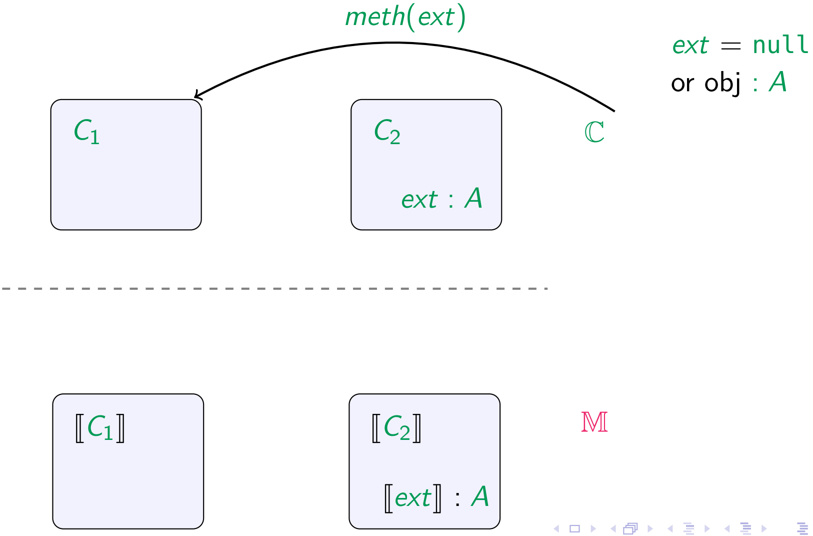
Object Faking



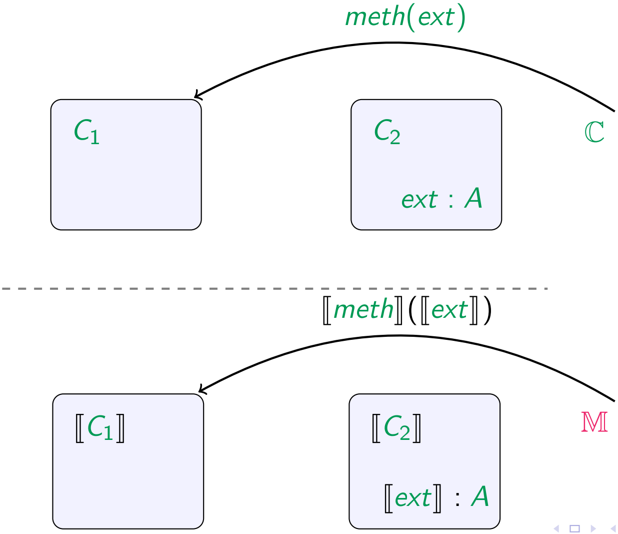
Object Faking



Object Faking

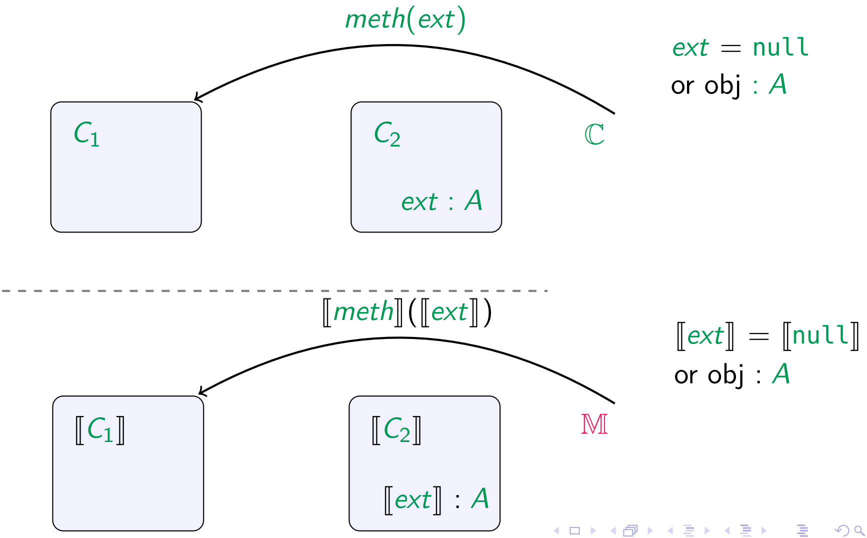


Object Faking

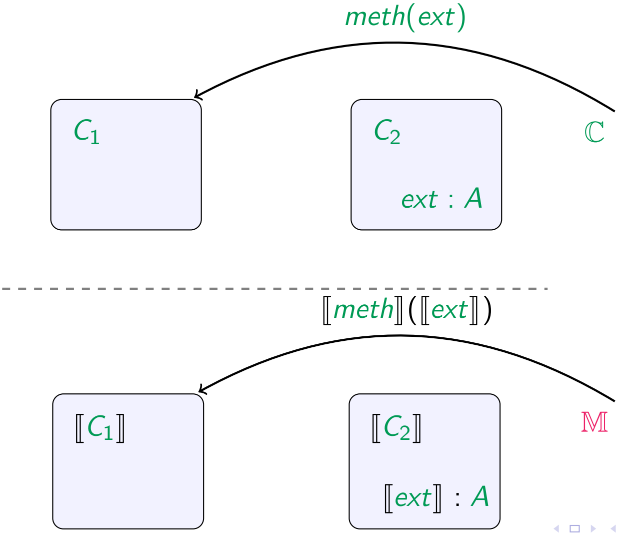


$ext = \texttt{null}$
or obj $: A$

Object Faking



Object Faking



$ext = \texttt{null}$
or obj : A

$[\![ext]\!] = [\![\texttt{null}]\!]$
or obj : A
or obj : $B \not\lessdot A$
or garbage

Motivation

- two (or more) components mean problems

Motivation

- two (or more) components mean problems
- cannot reuse existing secure compilers

Motivation

- two (or more) components mean problems
- cannot reuse existing secure compilers
  - generate code duplication

Motivation

- two (or more) components mean problems
- cannot reuse existing secure compilers
  - generate code duplication
  - require compiling whole codebase

Motivation

- two (or more) components mean problems
- cannot reuse existing secure compilers
  - generate code duplication
  - require compiling whole codebase
  - non-feasible since trust is not transitive

Solution Overview

- we devise $[\![\cdot]\!]^J_A$

Solution Overview

- we devise $[\![\cdot]\!]^J_A$
- $[\![\cdot]\!]^J_A$ places a *component C* into its own *PMA module*

Solution Overview

- we devise $[\![\cdot]\!]^{J}_{A}$
- $[\![\cdot]\!]^{J}_{A}$ places a *component* C into its own *PMA module*
- $[\![\cdot]\!]^{J}_{A}$ adds runtime checks to compiled components

Solution Overview

- we devise $[\![\cdot]\!]^J_A$
- $[\![\cdot]\!]^J_A$ places a *component* C into its own *PMA module*
- $[\![\cdot]\!]^J_A$ adds runtime checks to compiled components
- $[\![\cdot]\!]^J_A$ introduces a trusted module *Sys*

Solution Overview

- we devise $[\![\cdot]\!]^J_A$
- $[\![\cdot]\!]^J_A$ places a *component* C into its own *PMA module*
- $[\![\cdot]\!]^J_A$ adds runtime checks to compiled components
- $[\![\cdot]\!]^J_A$ introduces a trusted module *Sys*
- *Sys* tracks calls and object metadata

Solution Overview

- we devise $[\![\cdot]\!]^{J}_{A}$
- $[\![\cdot]\!]^{J}_{A}$ places a *component* C into its own *PMA module*
- $[\![\cdot]\!]^{J}_{A}$ adds runtime checks to compiled components
- $[\![\cdot]\!]^{J}_{A}$ introduces a trusted module *Sys*
- *Sys* tracks calls and object metadata

1. What is a PMA module?

Solution Overview

- we devise $[\![\cdot]\!]^{J}_{A}$
- $[\![\cdot]\!]^{J}_{A}$ places a *component* C into its own *PMA module*
- $[\![\cdot]\!]^{J}_{A}$ adds runtime checks to compiled components
- $[\![\cdot]\!]^{J}_{A}$ introduces a trusted module *Sys*
- *Sys* tracks calls and object metadata

1. What is a PMA module?
2. How does this address the previous problems?

What is PMA?

- deep encapsulation at hardware level

What is PMA?

- deep encapsulation at hardware level
- security building block of many security-relevant works

What is PMA?

- deep encapsulation at hardware level
- security building block of many security-relevant works
- readily available: Intel SGX

Untyped Assembly + PMA

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
⋮
0x0b52    movs r0 0x0b55
0x0b53    call 0x0002
0x0b54    movs r0 0xeb54
0x0b55    ...
⋮
0xab00    jmp 0x0b53
⋮
0xeb52    movs r0 0xeb54
0xeb53    call 0xab02
0xeb54    ...
⋮
```

Untyped Assembly + PMA

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
⋮
--- ID 1 ---
0x0b52    movs r0 0x0b55
0x0b53    call 0x0002
0x0b54    movs r0 0xeb54
0x0b55    ...
⋮
0xab00    jmp 0x0b53
⋮
--- ID 2 ---
0xeb52    movs r0 0xeb54
0xeb53    call 0xab02
0xeb54    ...
⋮
```

Untyped Assembly + PMA

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
  ⋮
```

ID 1

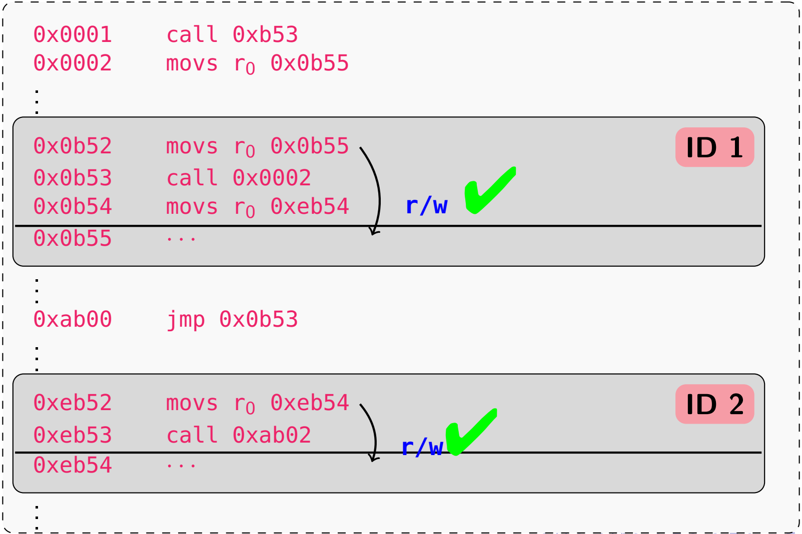
```
0x0b52    movs r0 0x0b55
0x0b53    call 0x0002
0x0b54    movs r0 0xeb54
-------------------------
0x0b55    ...
```

```
  ⋮
0xab00    jmp 0x0b53
  ⋮
```

ID 2

```
0xeb52    movs r0 0xeb54
0xeb53    call 0xab02
-------------------------
0xeb54    ...
  ⋮
```

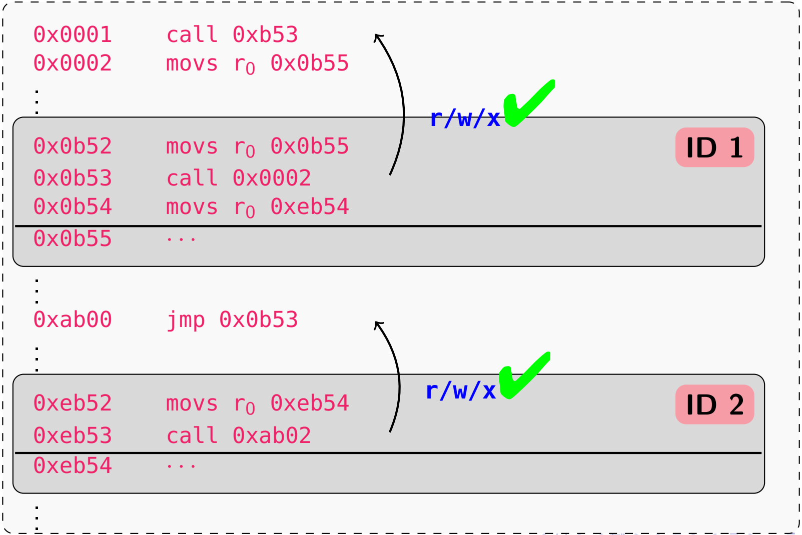
Untyped Assembly + PMA



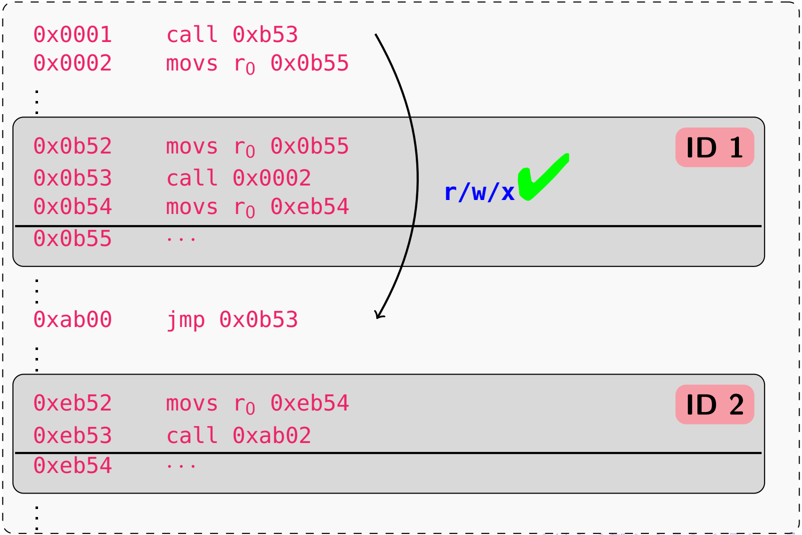
Untyped Assembly + PMA

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
  ⋮
ID 1
0x0b52    movs r0 0x0b55   ↷ r/x ✔
0x0b53    call 0x0002
0x0b54    movs r0 0xeb54
------------------------------
0x0b55    ...
  ⋮
0xab00    jmp 0x0b53
  ⋮
ID 2
0xeb52    movs r0 0xeb54
0xeb53    call 0xab02
------------------------------
0xeb54    ...
  ⋮
```

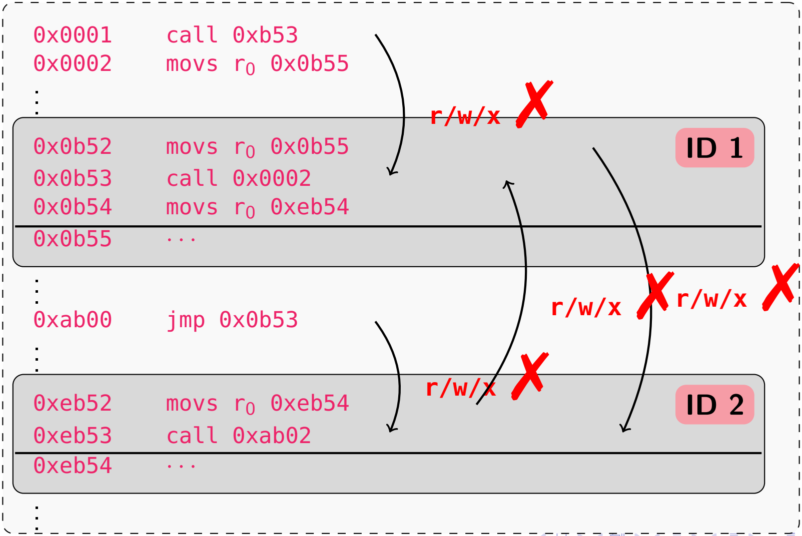
Untyped Assembly + PMA



Untyped Assembly + PMA



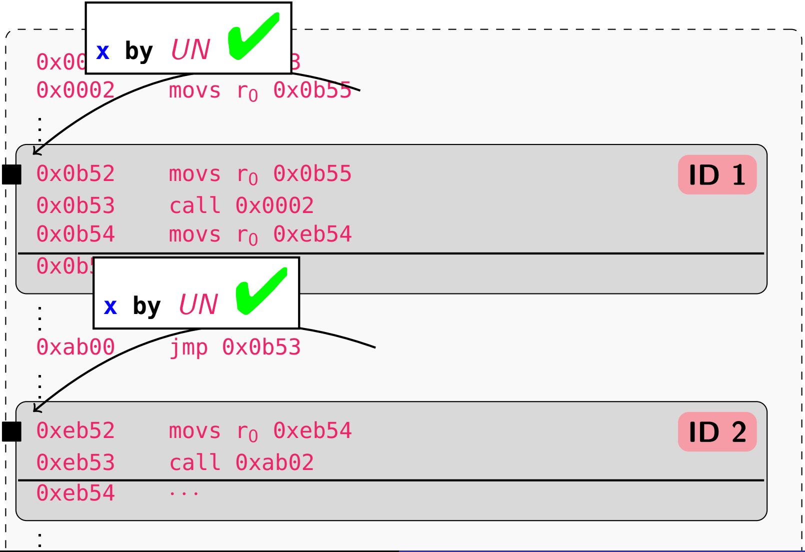
Untyped Assembly + PMA



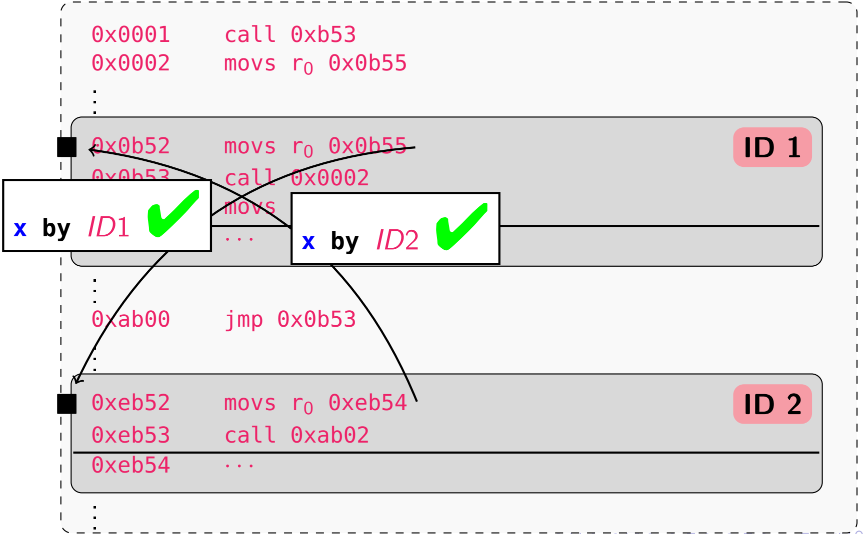
Untyped Assembly + PMA

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
  ⋮
0x0b52    movs r0 0x0b55      ID 1
0x0b53    call 0x0002
0x0b54    movs r0 0xeb54
---------------------------------
0x0b55    ...
  ⋮
0xab00    jmp 0x0b53
  ⋮
0xeb52    movs r0 0xeb54      ID 2
0xeb53    call 0xab02
---------------------------------
0xeb54    ...
  ⋮
```

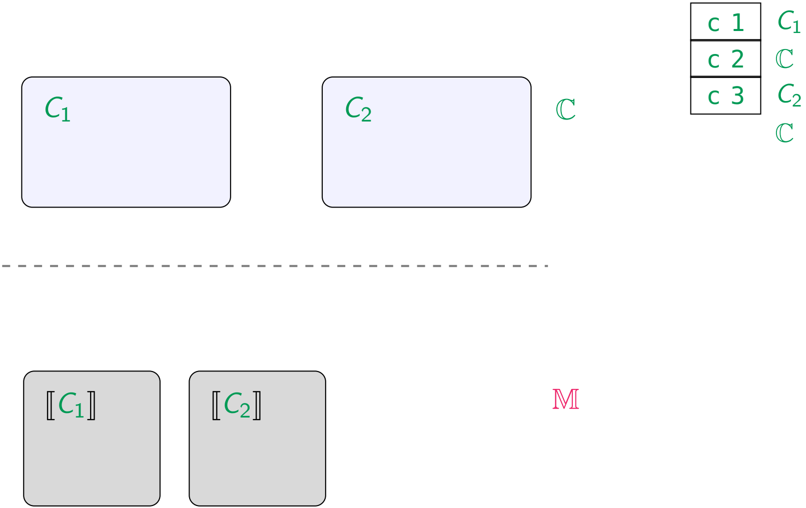
Untyped Assembly + PMA



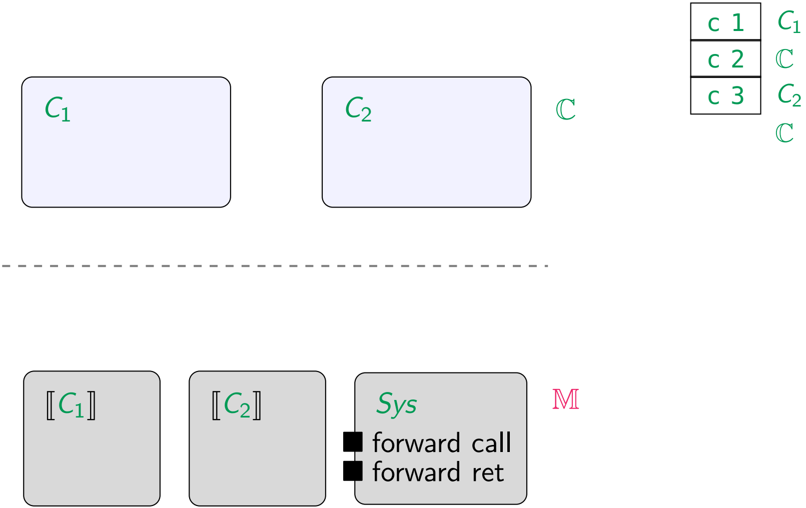
Untyped Assembly + PMA



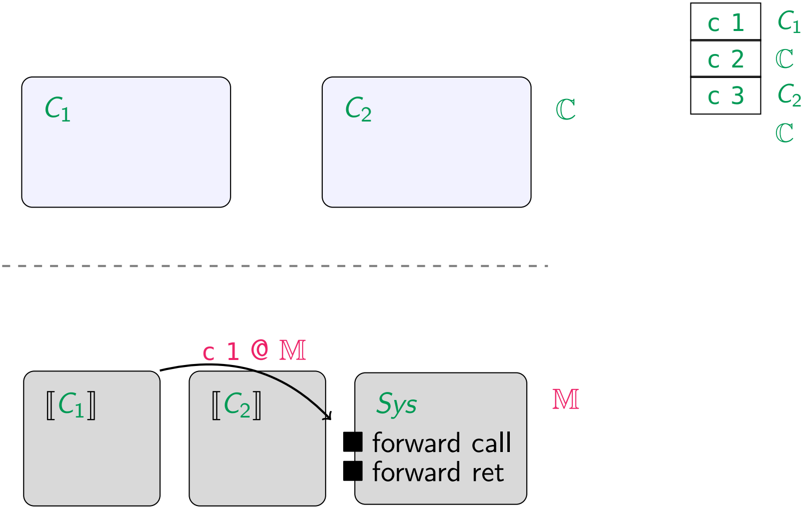
Addressing Call Stack Shortcutting



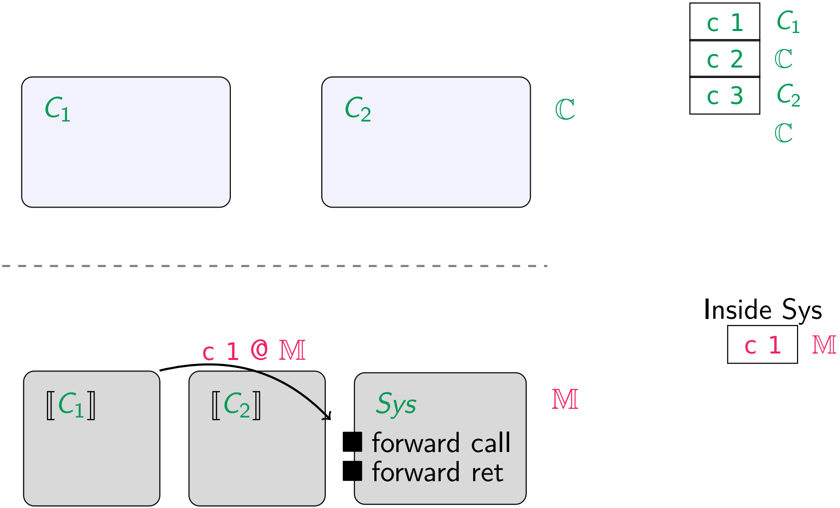
Addressing Call Stack Shortcutting



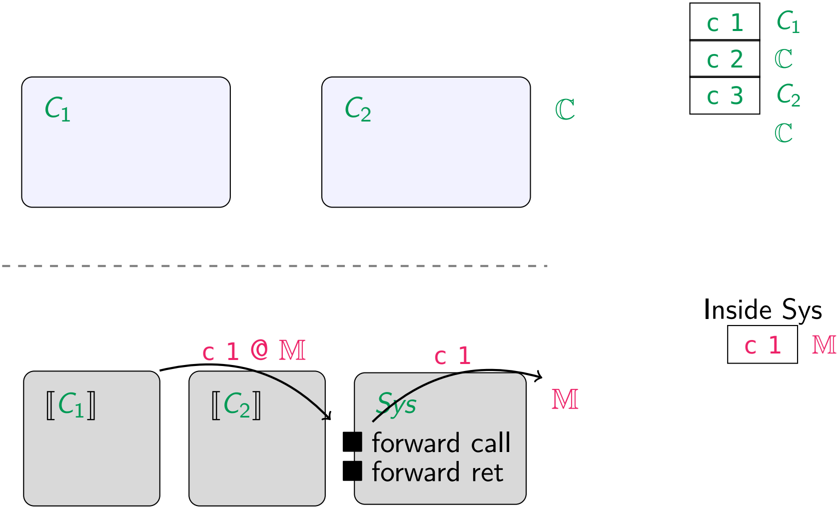
Addressing Call Stack Shortcutting



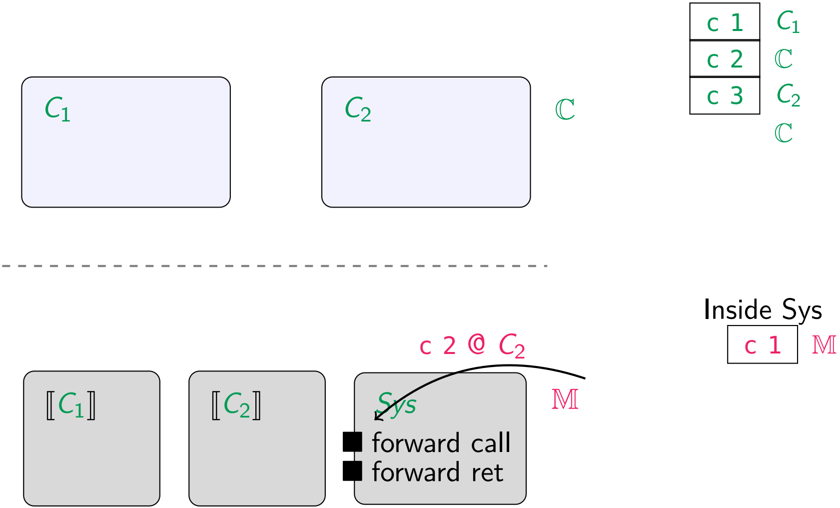
Addressing Call Stack Shortcutting



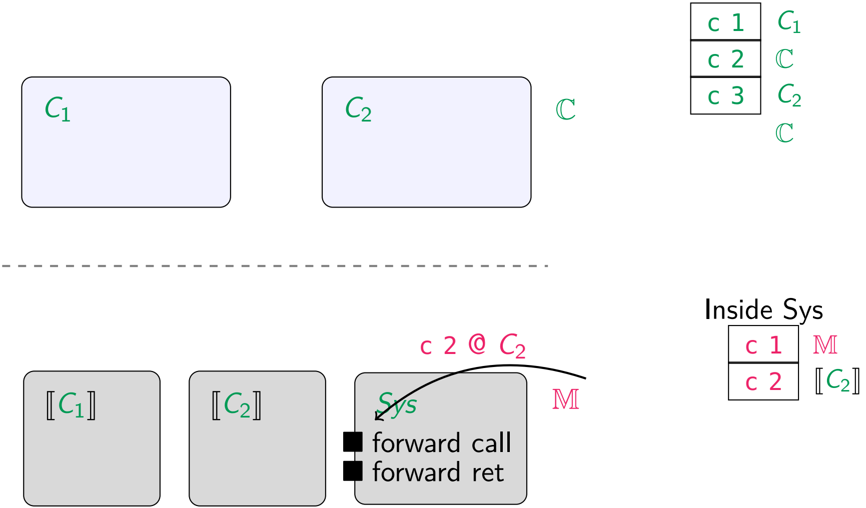
Addressing Call Stack Shortcutting



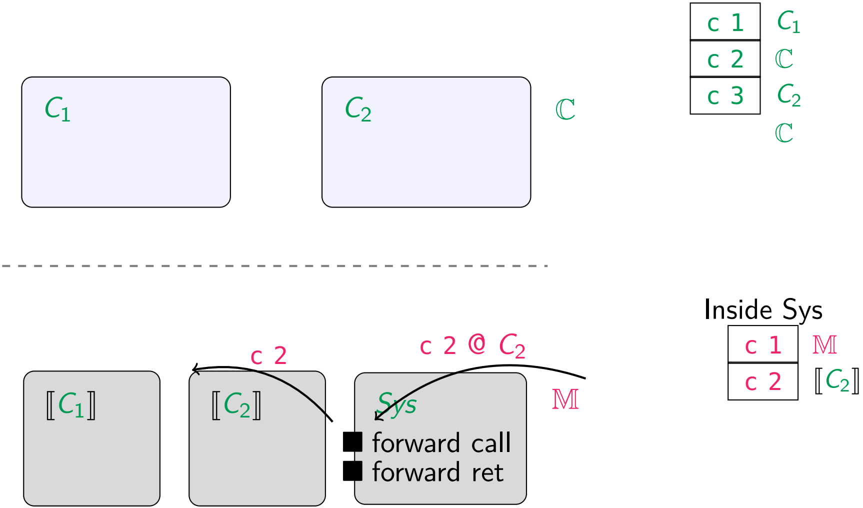
Addressing Call Stack Shortcutting



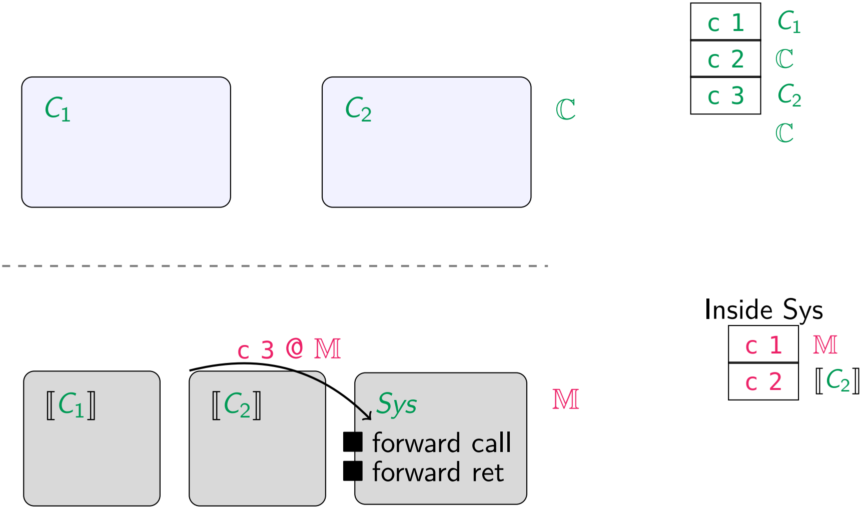
Addressing Call Stack Shortcutting



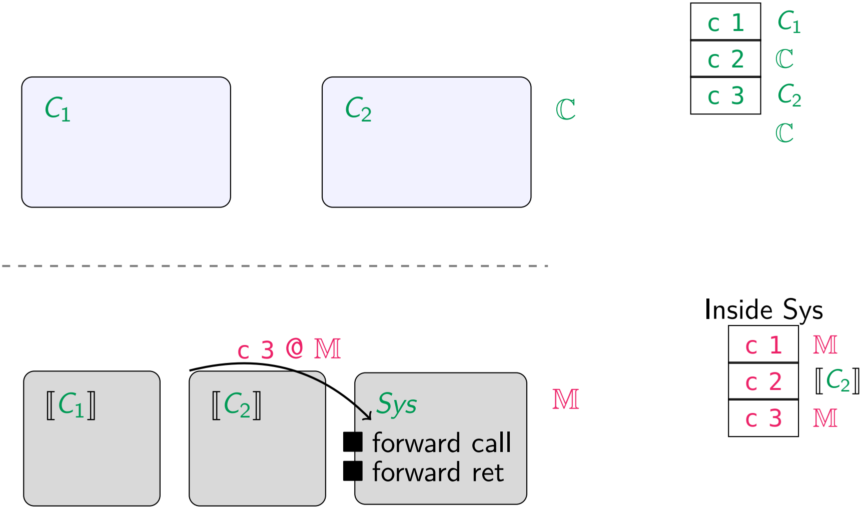
Addressing Call Stack Shortcutting



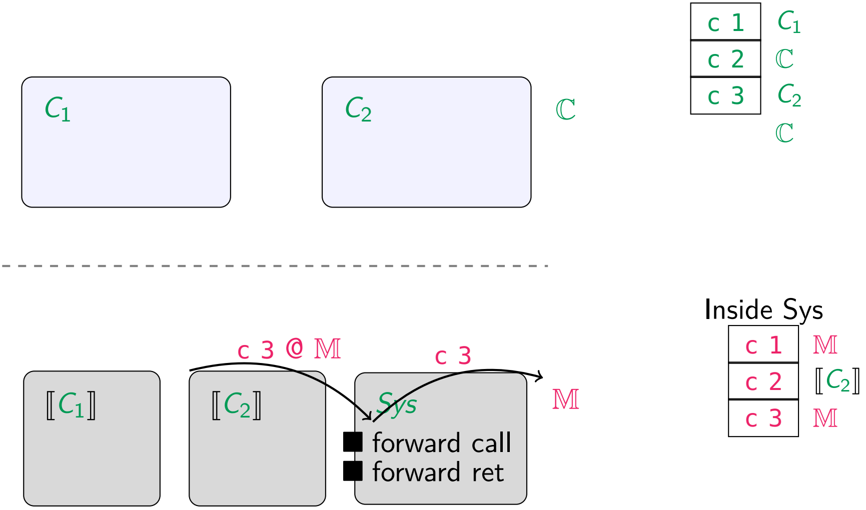
Addressing Call Stack Shortcutting



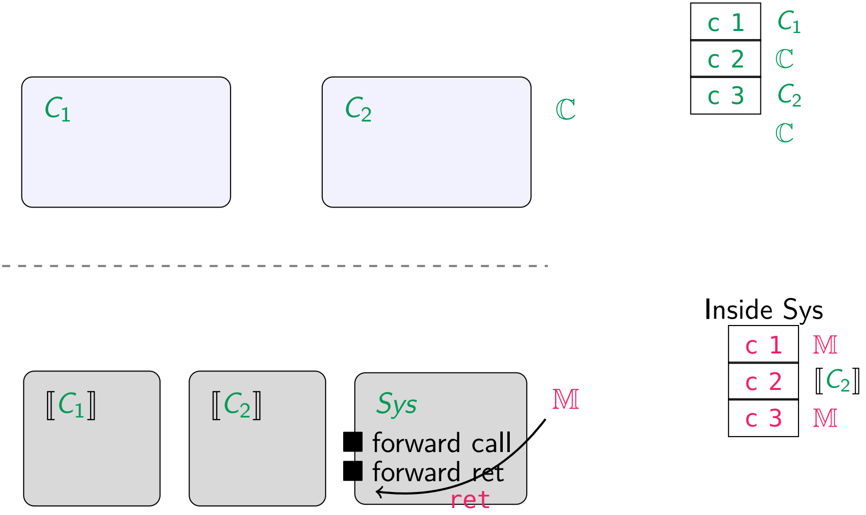
Addressing Call Stack Shortcutting



Addressing Call Stack Shortcutting

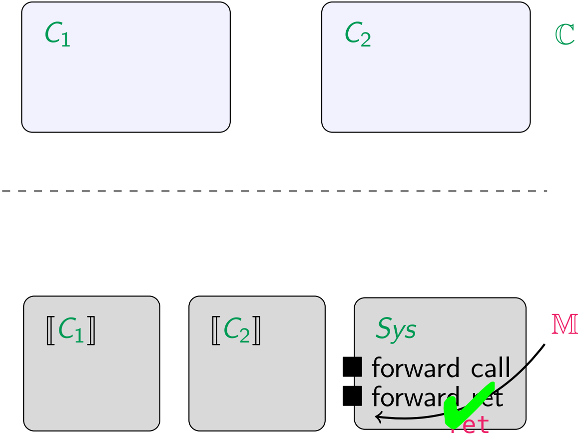


Addressing Call Stack Shortcutting



Addressing Call Stack Shortcutting

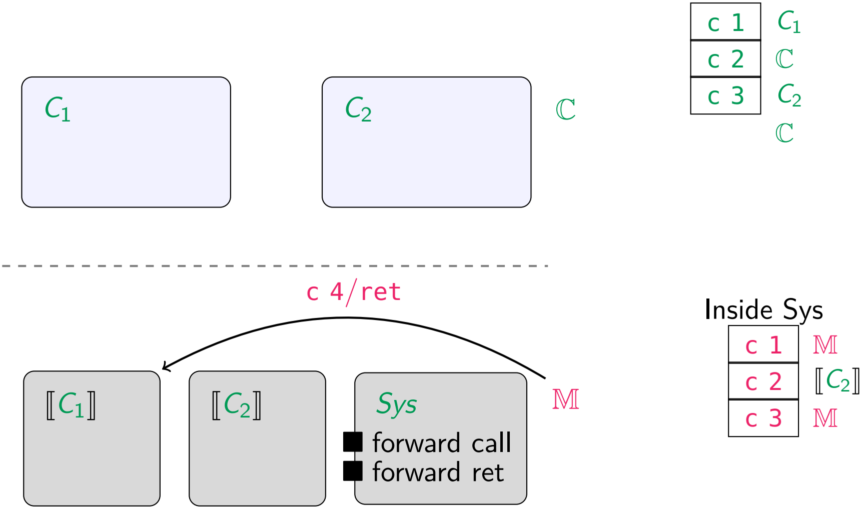
c 1	C_1
c 2	$\mathbb{C}$
c 3	C_2
	$\mathbb{C}$



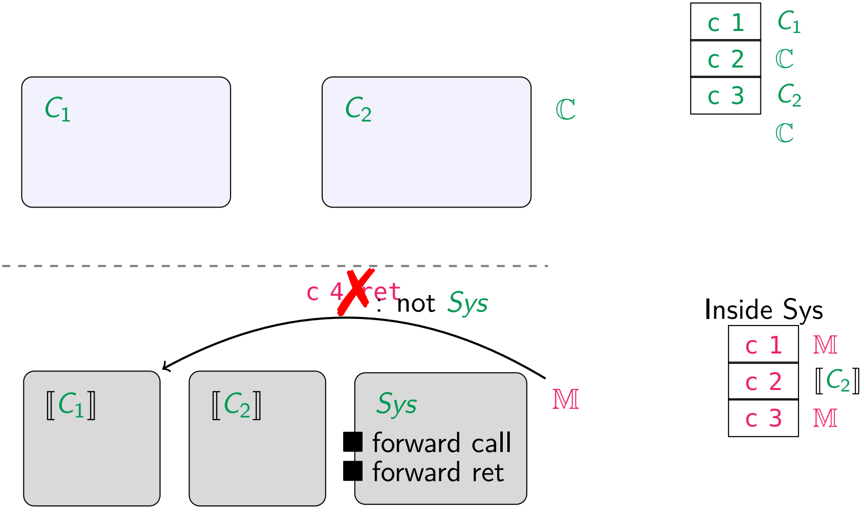
Inside Sys

c 1	$\mathbb{M}$
c 2	$[\![C_2]\!]$
c 3	$\mathbb{M}$

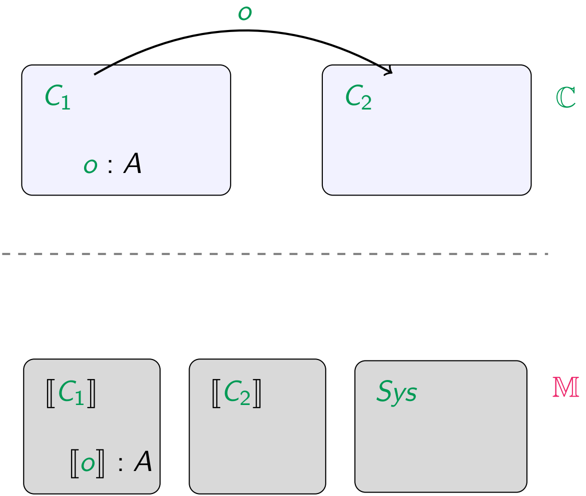
Addressing Call Stack Shortcutting



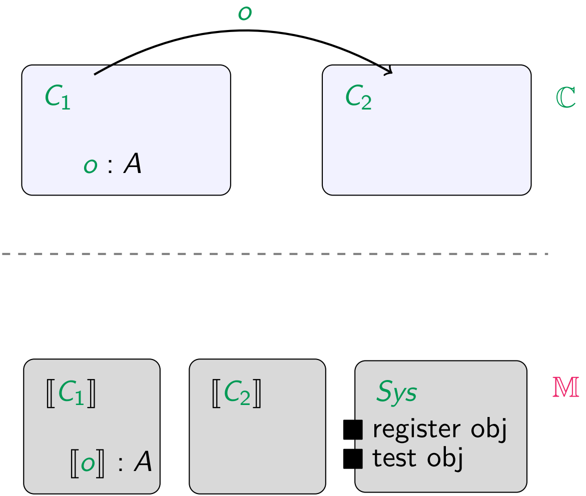
Addressing Call Stack Shortcutting



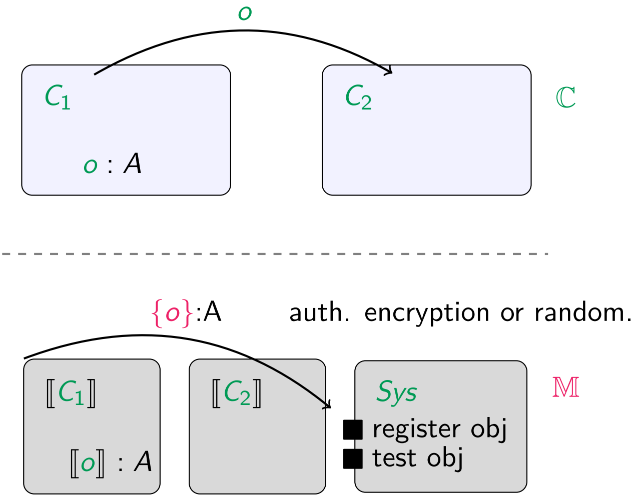
Addressing Object Guessing & Faking



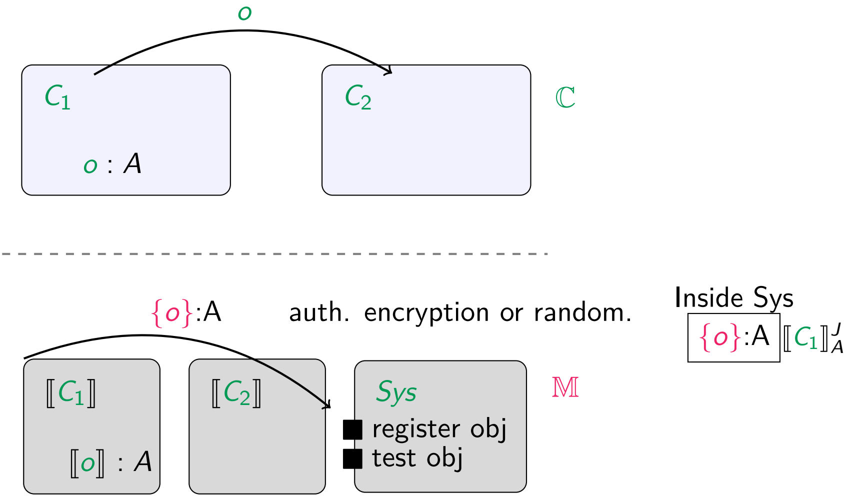
Addressing Object Guessing & Faking



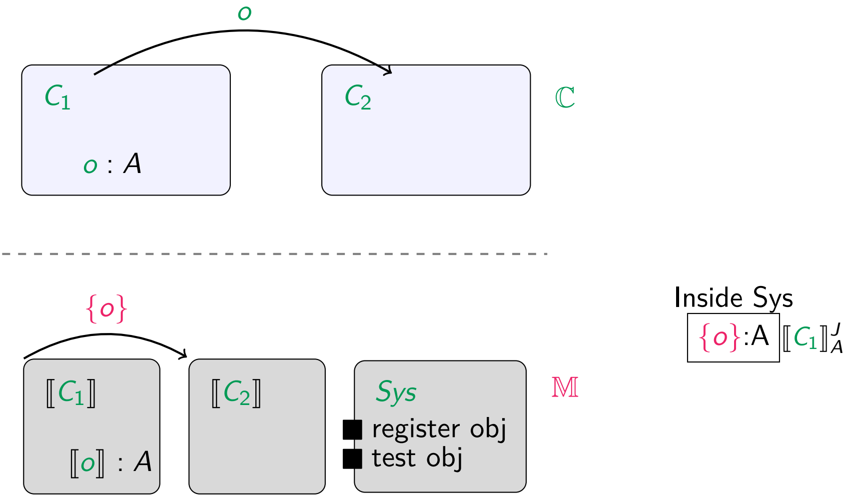
Addressing Object Guessing & Faking



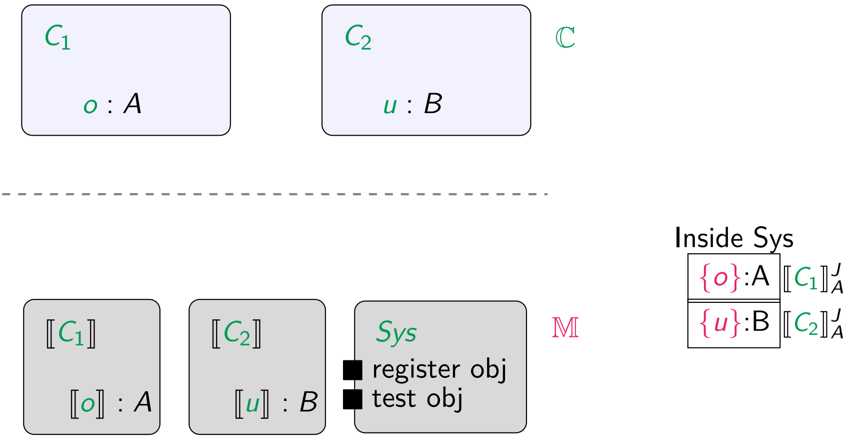
Addressing Object Guessing & Faking



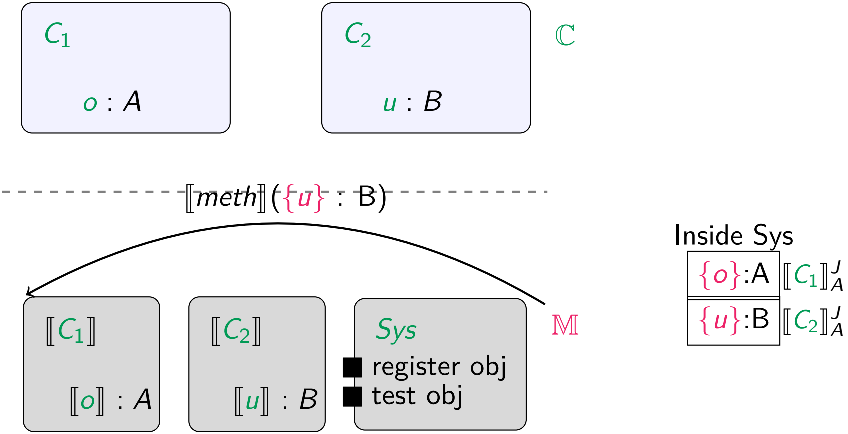
Addressing Object Guessing & Faking



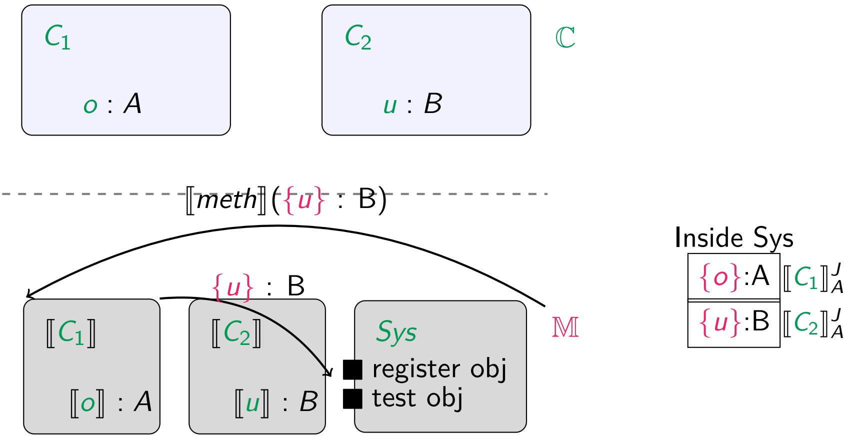
Addressing Object Guessing & Faking



Addressing Object Guessing & Faking



Addressing Object Guessing & Faking



More Solutions

- the paper describes more problems
- and how to address them

Formal Guarantees

- what tells us that $[\![\cdot]\!]^J_A$ is secure?

Formal Guarantees

- what tells us that $[\![\cdot]\!]^J_A$ is secure?
- $[\![\cdot]\!]^J_A$ is fully abstract (like others)

Formal Guarantees

- what tells us that $[\![\cdot]\!]^J_A$ is secure?
- $[\![\cdot]\!]^J_A$ is fully abstract (like others)
- $[\![\cdot]\!]^J_A$ **also** has *modular full-abstraction*

Formal Guarantees

- what tells us that $[\![\cdot]\!]^{J}_{A}$ is secure?
- $[\![\cdot]\!]^{J}_{A}$ is fully abstract (like others)
- $[\![\cdot]\!]^{J}_{A}$ **also** has *modular full-abstraction*
- no new machinery but the following is needed:
  - compiler modularity
  - compiler full-abstraction

Compiler Modular Full-Abstraction

$$\forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4.$$
$$\forall \mathcal{P}. [\![\mathcal{C}_2]\!]^J_A \simeq^{\mathcal{T}} \mathcal{P}$$
$$\forall \mathcal{P}'. [\![\mathcal{C}_4]\!]^J_A \simeq^{\mathcal{T}} \mathcal{P}'$$

$$\mathcal{C}_1 + \mathcal{C}_2 \simeq^{\mathcal{S}} \mathcal{C}_3 + \mathcal{C}_4$$
$$\Updownarrow$$
$$[\![\mathcal{C}_1]\!]^J_A + \mathcal{P} \simeq^{\mathcal{T}} [\![\mathcal{C}_3]\!]^J_A + \mathcal{P}'$$

Compiler Modular Full-Abstraction

$$\forall \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4.$$
$$\forall \mathcal{P}. [\![\mathcal{C}_2]\!]^J_A \simeq^{\mathcal{T}} \mathcal{P}$$
$$\forall \mathcal{P}'. [\![\mathcal{C}_4]\!]^J_A \simeq^{\mathcal{T}} \mathcal{P}'$$

$$\mathcal{C}_1 + \mathcal{C}_2 \simeq^{\mathcal{S}} \mathcal{C}_3 + \mathcal{C}_4$$
$$\Updownarrow$$
$$[\![\mathcal{C}_1]\!]^J_A + \mathcal{P} \simeq^{\mathcal{T}} [\![\mathcal{C}_3]\!]^J_A + \mathcal{P}'$$

- $\mathcal{P}$ and $\mathcal{P}'$ can even be hand-optimized
- as long as they behave like $[\![\mathcal{C}_2]\!]^J_A$ and $[\![\mathcal{C}_4]\!]^J_A$

Questions

Thank you!

Qs ?