

# Tales of Belgium: Reasoning about Capability Machines using Logical Relations

Lau Skorstengaard

Aarhus University

Aarhus University, December 2016

# Road map

Capability Machine

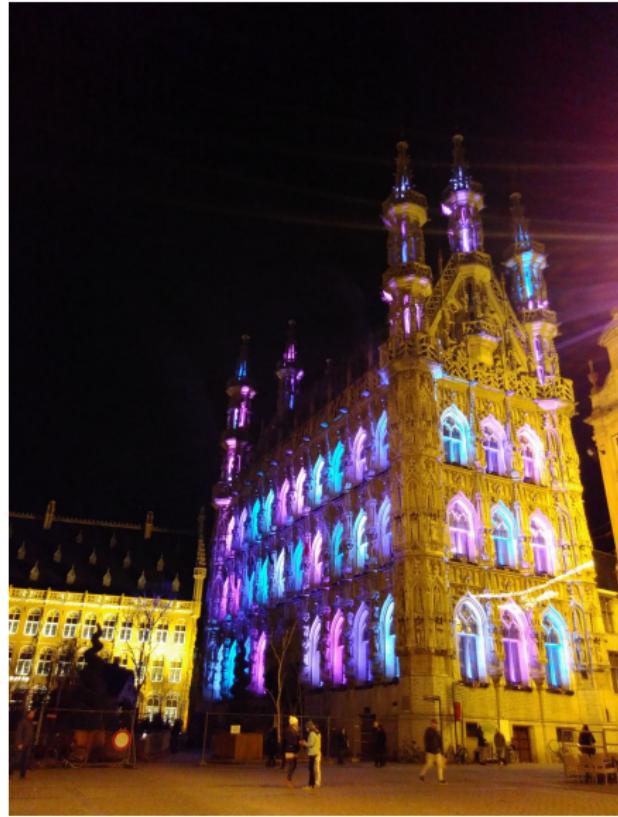
Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion

# Leuven



# Leuven

# Leuven



# Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion

# Why should I care about capability machines?

## **Current low-level protection mechanisms**

- ▶ Coarse-grained compartmentalisation
- ▶ Expensive context switches
- ▶ Well suited for high-level applications
- ▶ Does not scale well
- ▶ E.g., virtual memory

# Why should I care about capability machines?

## Capability machines

- ▶ Fine-grained compartmentalisation
- ▶ Cheap compartments
- ▶ Fine-grained sharing
- ▶ Well suited for applications with need for many compartments

# Capabilities

What is a capability?

# Capabilities

What is a capability?

- ▶ *Unforgeable token of authority*

# Capabilities

What is a capability?

- ▶ *Unforgeable token of authority*

What is a capability in a capability machine?

# Capabilities

What is a capability?

- ▶ *Unforgeable* token of authority

What is a capability in a capability machine?

- ▶ Unforgeable pointer

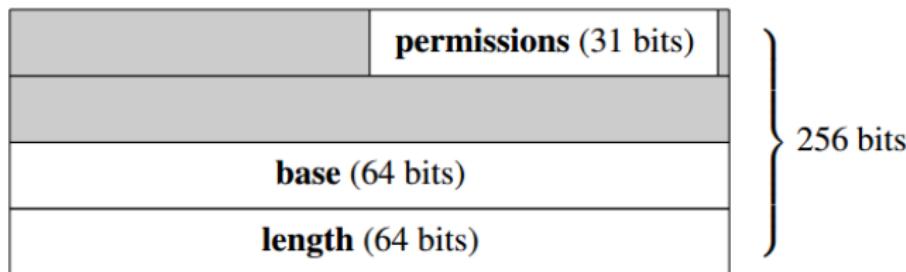


Figure: CHERI capability [1]

# Capabilities

What is a capability?

- ▶ *Unforgeable* token of authority

What is a capability in a capability machine?

- ▶ Unforgeable pointer
- ▶ Range of memory

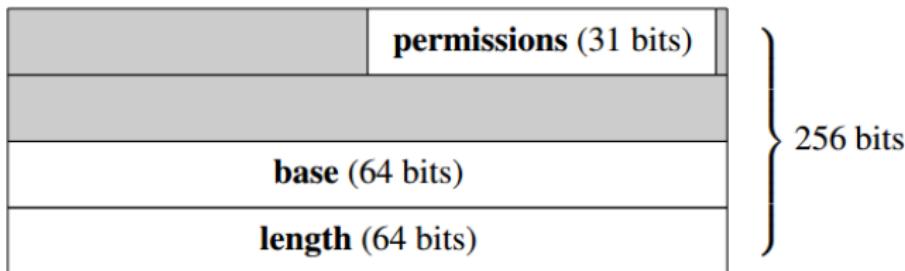


Figure: CHERI capability [1]

# Capabilities

What is a capability?

- ▶ *Unforgeable* token of authority

What is a capability in a capability machine?

- ▶ Unforgeable pointer
- ▶ Range of memory
- ▶ Permission

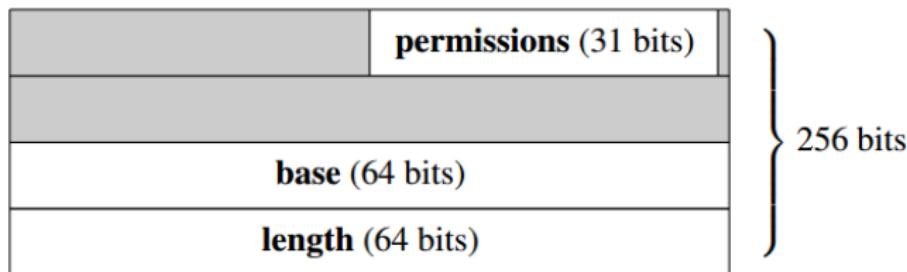


Figure: CHERI capability [1]

# Capability permissions

- ▶ Read
- ▶ Write
- ▶ Execute

# Capability permissions

- ▶ Read
- ▶ Write
- ▶ Execute
- ▶ Enter

# Capability permissions

- ▶ Read
- ▶ Write
- ▶ Execute
- ▶ Enter
  - ▶ When jumped to, it becomes a read and execute capability
  - ▶ Cannot be used in any other way

# Capability permissions

- ▶ Read
- ▶ Write
- ▶ Execute
- ▶ Enter
  - ▶ When jumped to, it becomes a read and execute capability
  - ▶ Cannot be used in any other way
  - ▶ Used by distrusting pieces of code to cross security boundaries

# Capability permissions

- ▶ Read
- ▶ Write
- ▶ Execute
- ▶ Enter
  - ▶ When jumped to, it becomes a read and execute capability
  - ▶ Cannot be used in any other way
  - ▶ Used by distrusting pieces of code to cross security boundaries
  - ▶ Modularisation

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine
  - ▶ jmp, jnz, move, plus, load, store

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine
  - ▶ *jmp, jnz, move, plus, load, store*
  - ▶ Instructions may require capability with certain permission.

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine
  - ▶ jmp, jnz, move, plus, load, store
  - ▶ Instructions may require capability with certain permission.
- ▶ Capability manipulation instructions

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine
  - ▶ `jmp`, `jnz`, `move`, `plus`, `load`, `store`
  - ▶ Instructions may require capability with certain permission.
- ▶ Capability manipulation instructions
  - ▶ `lea`, `restrict`, `subseg`

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine
  - ▶ `jmp`, `jnz`, `move`, `plus`, `load`, `store`
  - ▶ Instructions may require capability with certain permission.
- ▶ Capability manipulation instructions
  - ▶ `lea`, `restrict`, `subseg`
  - ▶ No instruction generates new capability

# Capability machine instructions

- ▶ Same instructions as in a normal low-level machine
  - ▶ `jmp`, `jnz`, `move`, `plus`, `load`, `store`
  - ▶ Instructions may require capability with certain permission.
- ▶ Capability manipulation instructions
  - ▶ `lea`, `restrict`, `subseg`
  - ▶ No instruction generates new capability
  - ▶ Manipulation of capabilities cannot result in authority amplification

# Capability machine overview

- ▶ Capabilities

# Capability machine overview

- ▶ Capabilities
  - ▶ Permissions

# Capability machine overview

- ▶ Capabilities
  - ▶ Permissions
  - ▶ Range of authority

# Capability machine overview

- ▶ Capabilities
  - ▶ Permissions
  - ▶ Range of authority
- ▶ Capability aware instructions

# Capability machine overview

- ▶ Capabilities
  - ▶ Permissions
  - ▶ Range of authority
- ▶ Capability aware instructions
- ▶ Memory and registers

# Capability machine overview

- ▶ Capabilities
  - ▶ Permissions
  - ▶ Range of authority
- ▶ Capability aware instructions
- ▶ Memory and registers
  - ▶ Can contain data and capabilities

# Capability machine overview

- ▶ Capabilities
  - ▶ Permissions
  - ▶ Range of authority
- ▶ Capability aware instructions
- ▶ Memory and registers
  - ▶ Can contain data and capabilities
  - ▶ Capabilities tagged

## Simple capability machine limitations

- ▶ No revocation of capabilities

## Simple capability machine limitations

- ▶ No revocation of capabilities
- ▶ Simulating revocation of enter capabilities:

## Simple capability machine limitations

- ▶ No revocation of capabilities
- ▶ Simulating revocation of enter capabilities:
  - ▶ On jump overwrite first address with fail instruction

## Simple capability machine limitations

- ▶ No revocation of capabilities
- ▶ Simulating revocation of enter capabilities:
  - ▶ On jump overwrite first address with fail instruction
  - ▶ Subsequent jumps fail

# Simple capability machine limitations

- ▶ No revocation of capabilities
- ▶ Simulating revocation of enter capabilities:
  - ▶ On jump overwrite first address with fail instruction
  - ▶ Subsequent jumps fail
- ▶ Issues:

# Simple capability machine limitations

- ▶ No revocation of capabilities
- ▶ Simulating revocation of enter capabilities:
  - ▶ On jump overwrite first address with fail instruction
  - ▶ Subsequent jumps fail
- ▶ Issues:
  - ▶ Memory leak

# Simple capability machine limitations

- ▶ No revocation of capabilities
- ▶ Simulating revocation of enter capabilities:
  - ▶ On jump overwrite first address with fail instruction
  - ▶ Subsequent jumps fail
- ▶ Issues:
  - ▶ Memory leak
  - ▶ Does not scale to other types of permissions

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:
  - ▶ Local/(global)

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:
  - ▶ Local/(global)
  - ▶ Permit write local (pwl)

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:
  - ▶ Local/(global)
  - ▶ Permit write local (pwl)
- ▶ Only pwl-capabilities can write local capabilities.

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:
  - ▶ Local/(global)
  - ▶ Permit write local (pwl)
- ▶ Only pwl-capabilities can write local capabilities.
- ▶ Gives simple temporal revocation, but

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:
  - ▶ Local/(global)
  - ▶ Permit write local (pwl)
- ▶ Only pwl-capabilities can write local capabilities.
- ▶ Gives simple temporal revocation, but
  - ▶ requires no global pwl-capabilities

## Local capabilities

- ▶ Idea: New type of capability that cannot be stored when “crossing security boundaries”
- ▶ Capabilities get two new fields:
  - ▶ Local/(global)
  - ▶ Permit write local (pwl)
- ▶ Only pwl-capabilities can write local capabilities.
- ▶ Gives simple temporal revocation, but
  - ▶ requires no global pwl-capabilities
  - ▶ enforcement depends on programming discipline.

# Brussels



# Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions

$$\text{Perm} \stackrel{\text{def}}{=} \{ \quad \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions,

$$\text{Perm} \stackrel{\text{def}}{=} \{o, \quad \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions, read only,

$$\text{Perm} \stackrel{\text{def}}{=} \{\text{o}, \text{ro}, \quad \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions, read only, read-write,

$$\text{Perm} \stackrel{\text{def}}{=} \{\text{o}, \text{ro}, \text{rw}, \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions, read only, read-write, read-execute,

$$\text{Perm} \stackrel{\text{def}}{=} \{ \text{o}, \text{ro}, \text{rw}, \quad \text{rx}, \quad \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions, read only, read-write, read-execute, enter,

$$\text{Perm} \stackrel{\text{def}}{=} \{ \text{o}, \text{ro}, \text{rw}, \quad \text{rx}, \text{e}, \quad \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions, read only, read-write, read-execute, enter, read-write-execute,

$$\text{Perm} \stackrel{\text{def}}{=} \{ \text{o}, \text{ro}, \text{rw}, \quad \text{rx}, \text{e}, \text{rwx}, \quad \}$$

# Formalisation - Permissions

## Permissions

- ▶ To simplify matters, we only allow certain combinations of permissions
- ▶ No permissions, read only, read-write, read-'write-local'  
read-execute, enter, read-write-execute,  
read-'write-local'-execute

$$\text{Perm} \stackrel{\text{def}}{=} \{\text{o}, \text{ro}, \text{rw}, \text{rwl}, \text{rx}, \text{e}, \text{rwx}, \text{rwlx}\}$$

# Formalisation - Permissions

## Permissions ordering

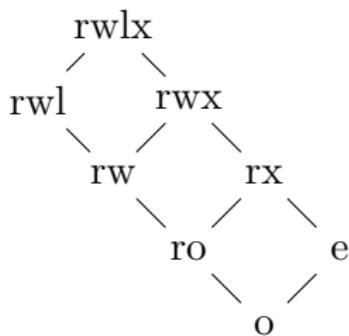


Figure: Permission hierarchy

# Formalisation - Locality

## Locality

Global ::= {global, local}

## Locality ordering



Figure: Locality hierarchy

# Formalisation - Capabilities

## Capability

$\text{Cap} \stackrel{\text{\tiny def}}{=}$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality

$$\text{Cap} \stackrel{\text{\tiny def}}{=}$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global})$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality
- ▶ Range of authority

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global})$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality
- ▶ Range of authority

$$\text{Addr} \stackrel{\text{\tiny def}}{=} \mathbb{N}$$

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global})$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality
- ▶ Range of authority

$$\text{Addr} \stackrel{\text{\tiny def}}{=} \mathbb{N}$$

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr}$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality
- ▶ Range of authority
- ▶ Pointer

$$\text{Addr} \stackrel{\text{\tiny def}}{=} \mathbb{N}$$

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr}$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality
- ▶ Range of authority
- ▶ Pointer

$$\text{Addr} \stackrel{\text{\tiny def}}{=} \mathbb{N}$$

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \times \text{Addr}$$

# Formalisation - Capabilities

## Capability

- ▶ Permission and locality
- ▶ Range of authority
- ▶ Pointer

$$\text{Addr} \stackrel{\text{\tiny def}}{=} \mathbb{N}$$

$$\text{Cap} \stackrel{\text{\tiny def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times \text{Addr} \times \text{Addr}$$

Example: ((e, local), 30, 42, 30)

# Formalisation - Words and register file

## Words

Word  $\stackrel{\text{def}}{=}$

# Formalisation - Words and register file

## Words

- ▶ Capability

Word  $\stackrel{\text{def}}{=}$

# Formalisation - Words and register file

## Words

- ▶ Capability

$$\text{Word} \stackrel{\text{def}}{=} \text{Cap}$$

# Formalisation - Words and register file

## Words

- ▶ Capability
- ▶ Data (and instructions)

Word  $\stackrel{\text{def}}{=}$  Cap

# Formalisation - Words and register file

## Words

- ▶ Capability
- ▶ Data (and instructions)

$$\text{Word} \stackrel{\text{def}}{=} \text{Cap} + \mathbb{Z}$$

# Formalisation - Words and register file

## Words

- ▶ Capability
- ▶ Data (and instructions)
- ▶ In the real machine capabilities are tagged

$$\text{Word} \stackrel{\text{def}}{=} \text{Cap} + \mathbb{Z}$$

# Formalisation - Words and register file

## Words

- ▶ Capability
- ▶ Data (and instructions)
- ▶ In the real machine capabilities are tagged

$$\text{Word} \stackrel{\text{def}}{=} \text{Cap} + \mathbb{Z}$$

## Register file

$$\text{Reg} \stackrel{\text{def}}{=}$$

# Formalisation - Words and register file

## Words

- ▶ Capability
- ▶ Data (and instructions)
- ▶ In the real machine capabilities are tagged

$$\text{Word} \stackrel{\text{def}}{=} \text{Cap} + \mathbb{Z}$$

## Register file

- ▶ Assume finite set of registers  $\text{RegisterName} \ni \text{pc}$

$$\text{Reg} \stackrel{\text{def}}{=}$$

# Formalisation - Words and register file

## Words

- ▶ Capability
- ▶ Data (and instructions)
- ▶ In the real machine capabilities are tagged

$$\text{Word} \stackrel{\text{def}}{=} \text{Cap} + \mathbb{Z}$$

## Register file

- ▶ Assume finite set of registers  $\text{RegisterName} \ni \text{pc}$

$$\text{Reg} \stackrel{\text{def}}{=} \text{RegisterName} \rightarrow \text{Word}$$

# Formalisation - Memory and configurations

## Memory

$\text{Mem} \stackrel{\text{def}}{=}$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

## Configuration

$$\text{Conf} \stackrel{\text{def}}{=}$$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

## Configuration

- ▶ Executable configuration

$$\text{Conf} \stackrel{\text{def}}{=}$$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

## Configuration

- ▶ Executable configuration

$$\text{Conf} \stackrel{\text{def}}{=} \text{Reg} \times \text{Mem}$$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

## Configuration

- ▶ Executable configuration
- ▶ Successfully halted configuration

$$\text{Conf} \stackrel{\text{def}}{=} \text{Reg} \times \text{Mem}$$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

## Configuration

- ▶ Executable configuration
- ▶ Successfully halted configuration

$$\text{Conf} \stackrel{\text{def}}{=} \text{Reg} \times \text{Mem} + \{\text{halted}\} \times \text{Mem}$$

# Formalisation - Memory and configurations

## Memory

- ▶ Map from Addr to Word

$$\text{Mem} \stackrel{\text{def}}{=} \text{Addr} \rightarrow \text{Word}$$

## Configuration

- ▶ Executable configuration
- ▶ Successfully halted configuration
- ▶ Failed configuration

$$\text{Conf} \stackrel{\text{def}}{=} \text{Reg} \times \text{Mem} + \{\text{failed}\} + \{\text{halted}\} \times \text{Mem}$$

# Formalisation - Instructions

## Syntax

Instructions ::=

# Formalisation - Instructions

## Syntax

$rv ::= n \mid r$

Instructions ::=

# Formalisation - Instructions

## Syntax

- ▶ The normal instructions

$rv ::= n \mid r$

Instructions ::=

# Formalisation - Instructions

## Syntax

- ▶ The normal instructions

$rv ::= n \mid r$

Instructions  $::= \text{jmp } r \mid \text{jnz } r \ rv \mid \text{move } r \ rv \mid$   
 $\text{load } r \ r \mid \text{store } r \ r \mid \text{plus } r \ rv \ rv$

# Formalisation - Instructions

## Syntax

- ▶ The normal instructions
- ▶ The capability manipulation instructions

$$\begin{array}{lcl} rv & ::= & n \mid r \\ \text{Instructions} & ::= & \text{jmp } r \mid \text{jnz } r \text{ } rv \mid \text{move } r \text{ } rv \mid \\ & & \text{load } r \text{ } r \mid \text{store } r \text{ } r \mid \text{plus } r \text{ } rv \text{ } rv \end{array}$$

# Formalisation - Instructions

## Syntax

- ▶ The normal instructions
- ▶ The capability manipulation instructions

```
rv ::= n | r
Instructions ::= jmp r | jnz r rv | move r rv |
                load r r | store r r | plus r rv rv |
                lea r rv | restrict r r rv |
                subseg r rv rv |
                getp r r | getl r r | getb r r |
                gete r r | geta r r
```

# Formalisation - Instructions

## Syntax

- ▶ The normal instructions
- ▶ The capability manipulation instructions
- ▶ Instructions to stop the machine

$rv ::= n \mid r$

Instructions  $::= jmp\ r \mid jnz\ r\ rv \mid move\ r\ rv \mid$   
 $load\ r\ r \mid store\ r\ r \mid plus\ r\ rv\ rv \mid$   
 $lea\ r\ rv \mid restrict\ r\ r\ rv \mid$   
 $subseg\ r\ rv\ rv \mid$   
 $getp\ r\ r \mid getl\ r\ r \mid getb\ r\ r \mid$   
 $gete\ r\ r \mid geta\ r\ r$

# Formalisation - Instructions

## Syntax

- ▶ The normal instructions
- ▶ The capability manipulation instructions
- ▶ Instructions to stop the machine

$rv ::= n \mid r$

Instructions  $::= jmp\ r \mid jnz\ r\ rv \mid move\ r\ rv \mid$   
 $load\ r\ r \mid store\ r\ r \mid plus\ r\ rv\ rv \mid$   
 $lea\ r\ rv \mid restrict\ r\ r\ rv \mid$   
 $subseg\ r\ rv\ rv \mid$   
 $getp\ r\ r \mid getl\ r\ r \mid getb\ r\ r \mid$   
 $gete\ r\ r \mid geta\ r\ r \mid fail \mid halt$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

---

*executionAllowed*( $\Phi$ )

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\Phi.\text{reg}(\text{pc}) = ((\textit{perm}, g), \textit{base}, \textit{end}, a)$$

---

*executionAllowed*( $\Phi$ )

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end}}{\text{executionAllowed}(\Phi)}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}}{\text{executionAllowed}(\Phi)}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}}{\text{executionAllowed}(\Phi)}$$

$$\frac{\neg \text{executionAllowed}(\Phi)}{\Phi \rightarrow}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}}{\text{executionAllowed}(\Phi)}$$

$$\frac{\neg \text{executionAllowed}(\Phi)}{\Phi \rightarrow \text{failed}}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}}{\text{executionAllowed}(\Phi)}$$

$$\frac{\neg \text{executionAllowed}(\Phi)}{\Phi \rightarrow \text{failed}}$$

$$\frac{\text{executionAllowed}(\Phi)}{\Phi \rightarrow}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}}{\text{executionAllowed}(\Phi)}$$

$$\frac{\neg \text{executionAllowed}(\Phi)}{\Phi \rightarrow \text{failed}}$$

$$\frac{\text{executionAllowed}(\Phi) \quad i = \Phi.\text{mem}(a)}{\Phi \rightarrow i}$$

# Formalisation - Operational Semantics (1)

## Execution relation

$$\rightarrow \subseteq (\text{Reg} \times \text{Mem}) \times \text{Conf}$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}, \text{rwlx}\}}{\text{executionAllowed}(\Phi)}$$

$$\frac{\neg \text{executionAllowed}(\Phi)}{\Phi \rightarrow \text{failed}}$$

$$\frac{\text{executionAllowed}(\Phi) \quad i = \Phi.\text{mem}(a)}{\Phi \rightarrow \llbracket i \rrbracket(\Phi)}$$

# Formalisation - Operational Semantics (2)

---

$\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) =$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2)$$

---

$$\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = \quad \quad \quad \Phi[\text{mem}.r_1 \mapsto w]$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a)$$

---

$$\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = \Phi[\text{mem.}a \mapsto w]$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket (\Phi) = \quad \quad \quad \Phi[\text{mem}.a \mapsto w]$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket (\Phi) = \quad \quad \quad \Phi[\text{mem.}a \mapsto w]$$

## Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((perm, g), base, end, a) \\ perm \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\ base \leq a \leq end \quad w = ((\_, \text{local}), \_, \_, \_) \end{array}}{\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = \quad \quad \quad \Phi[\text{mem}.a \mapsto w]}$$

---

## Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((perm, g), base, end, a) \\ perm \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\ base \leq a \leq end \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow perm \in \{\text{rwl}, \text{rwlx}\}}{\llbracket \text{store } r_1 \ r_2 \rrbracket(\Phi) = \Phi[\text{mem}.a \mapsto w]}$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$[\![\text{store } r_1 \; r_2]\!] (\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket (\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

---

$$\text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto \boxed{\quad}]$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a)$$

$$\text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket (\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

$$\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a)$$

---

$$\text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto ]$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket (\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

$$\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \\ \frac{}{\text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto \text{newPc}]}$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a)$$

$$\text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket (\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

$$\frac{\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \quad \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1)}{\text{updatePc}(\Phi) = \Phi[\text{reg.pc} \mapsto \text{newPc}]}$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket(\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

---

$$\llbracket \text{restrict } r_1 \; r_2 \; r_3 \rrbracket = \Phi[\text{reg}.r_1 \mapsto c]$$

$$\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \\ \hline \text{updatePc}(\Phi) = \Phi[\text{reg}.pc \mapsto \text{newPc}]$$

## Formalisation - Operational Semantics (2)

$$w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{perm} \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\}$$

$$\text{base} \leq a \leq \text{end} \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow \text{perm} \in \{\text{rwl}, \text{rwlx}\}$$

---

$$\llbracket \text{store } r_1 \; r_2 \rrbracket(\Phi) = \text{updatePc}(\Phi[\text{mem}.a \mapsto w])$$

$$\Phi.\text{reg}(r_2) = (\text{permPair}, \text{base}, \text{end}, a)$$

---

$$\llbracket \text{restrict } r_1 \; r_2 \; r_3 \rrbracket = \Phi[\text{reg}.r_1 \mapsto c]$$

$$\Phi.\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1)$$

---

$$\text{updatePc}(\Phi) = \Phi[\text{reg}.pc \mapsto \text{newPc}]$$

## Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((perm, g), base, end, a) \\ perm \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\ base \leq a \leq end \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow perm \in \{\text{rwl}, \text{rwlx}\} \end{array}}{\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = updatePc(\Phi[\text{mem}.a \mapsto w])}$$

$$\begin{array}{l} \Phi.\text{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \end{array}$$

---

$$\llbracket \text{restrict } r_1 \ r_2 \ r_3 \rrbracket = \Phi[\text{reg}.r_1 \mapsto c]$$

$$\frac{\begin{array}{c} \Phi.\text{reg}(\text{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\text{reg}.pc \mapsto newPc]}$$

## Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((perm, g), base, end, a) \\ perm \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\ base \leq a \leq end \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow perm \in \{\text{rwl}, \text{rwlx}\} \end{array}}{\llbracket \text{store } r_1 \ r_2 \rrbracket (\Phi) = updatePc(\Phi[\text{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\text{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \\ newPermPair \sqsubseteq permPair \end{array}}{\llbracket \text{restrict } r_1 \ r_2 \ r_3 \rrbracket = \Phi[\text{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\text{reg}(\text{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\text{reg}.pc \mapsto newPc]}$$

## Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((perm, g), base, end, a) \\ perm \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\ base \leq a \leq end \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow perm \in \{\text{rwl}, \text{rwlx}\} \end{array}}{\llbracket \text{store } r_1 \ r_2 \rrbracket(\Phi) = updatePc(\Phi[\text{mem}.a \mapsto w])}$$

$$\frac{\begin{array}{c} \Phi.\text{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \\ newPermPair \sqsubseteq permPair \quad c = (newPermPair, base, end, a) \end{array}}{\llbracket \text{restrict } r_1 \ r_2 \ r_3 \rrbracket = \Phi[\text{reg}.r_1 \mapsto c]}$$

$$\frac{\begin{array}{c} \Phi.\text{reg}(\text{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\text{reg}.pc \mapsto newPc]}$$

## Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{c} w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((perm, g), base, end, a) \\ perm \in \{\text{rw}, \text{rwl}, \text{rwx}, \text{rwlx}\} \\ base \leq a \leq end \quad w = ((\_, \text{local}), \_, \_, \_) \Rightarrow perm \in \{\text{rwl}, \text{rwlx}\} \end{array}}{\llbracket \text{store } r_1 \ r_2 \rrbracket(\Phi) = updatePc(\Phi[\text{mem}.a \mapsto w])}$$
$$\frac{\begin{array}{c} \Phi.\text{reg}(r_2) = (permPair, base, end, a) \\ newPermPair = decodePermPair(\Phi, r_3) \\ newPermPair \sqsubseteq permPair \quad c = (newPermPair, base, end, a) \end{array}}{\llbracket \text{restrict } r_1 \ r_2 \ r_3 \rrbracket = updatePc(\Phi[\text{reg}.r_1 \mapsto c])}$$

$$\frac{\begin{array}{c} \Phi.\text{reg}(\text{pc}) = ((perm, g), base, end, a) \\ newPc = (perm, base, end, a + 1) \end{array}}{updatePc(\Phi) = \Phi[\text{reg}.pc \mapsto newPc]}$$

## Formalisation - Operational Semantics (3)

- ▶ Need a *failed* case for each of the rules

## Formalisation - Operational Semantics (3)

- ▶ Need a *failed* case for each of the rules
- ▶ The operational semantics of the remaining instructions is defined in a similar fashion

# Antwerp



# Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion

## The “awkward” example

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
  assert(x == 1)
```

## The “awkward” example

```
g = fun _ =>
    let x = 0 in
    fun f =>
        x := 0;
        f();
        x := 1;
        f();
        assert(x == 1)
```

- ▶ Show for any reasonable `adv` that the assertion never fails for `adv(g)`.

## The “awkward” example

```
g = fun _ =>
    let x = 0 in
    fun f =>
        x := 0;
        f();
        x := 1;
        f();
        assert(x == 1)
```

- ▶ Show for any reasonable `adv` that the assertion never fails for `adv(g)`.
- ▶ Need to define some macros to make a readable translation.

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$`

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert  $rv_1$   $rv_2$`

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert  $rv_1$   $rv_2$` 
  - ▶ check whether  $rv_1$  and  $rv_2$  contains the same value

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert  $rv_1$   $rv_2$` 
  - ▶ check whether  $rv_1$  and  $rv_2$  contains the same value
    - ▶ if so: continue execution

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert  $rv_1$   $rv_2$` 
  - ▶ check whether  $rv_1$  and  $rv_2$  contains the same value
    - ▶ if so: continue execution
    - ▶ if not: set assertion flag and halt

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert  $rv_1 rv_2$` 
  - ▶ check whether  $rv_1$  and  $rv_2$  contains the same value
    - ▶ if so: continue execution
    - ▶ if not: set assertion flag and halt
- ▶ `malloc  $r n$`

# Macros

- ▶ `crtcls [( $x_1, r_1$ ), ..., ( $x_n, r_n$ )]  $r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert  $rv_1 rv_2$` 
  - ▶ check whether  $rv_1$  and  $rv_2$  contains the same value
    - ▶ if so: continue execution
    - ▶ if not: set assertion flag and halt
- ▶ `malloc  $r n$` 
  - ▶ allocates a *fresh* piece of memory of size  $n$

# Macros

- ▶ `crtcls  $[(x_1, r_1), \dots, (x_n, r_n)] r$` 
  - ▶ creates closure
  - ▶  $x_1, \dots, x_n$  available in program
  - ▶  $r$  capability for program
- ▶ `assert rv1 rv2`
  - ▶ check whether  $rv_1$  and  $rv_2$  contains the same value
    - ▶ if so: continue execution
    - ▶ if not: set assertion flag and halt
- ▶ `malloc r n`
  - ▶ allocates a *fresh* piece of memory of size  $n$
  - ▶ leaves a global capability with rwx permission in register  $r$

## The “awkward” example (naive translation)

```
g = fun _ =>
  let x = 0 in
    fun f =>
      x := 0;
      f();
      x := 1;
      f();
      assert(x == 1)
```

## The “awkward” example (naive translation)

```
g = fun _ =>
    let x = 0 in
        fun f =>
            x := 0;
            f();
            x := 1;
            f();
            assert(x == 1)
```

---

```
g :   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      jmp r0
```

## The “awkward” example (naive translation)

<pre>g = fun _ =&gt;     let x = 0 in         fun f =&gt;             x := 0;             f();             x := 1;             f();             assert(x == 1)</pre>	<hr/>	<pre>f :      store x 0                 jmp r1                 store x 1                 jmp r1                 load r1 x                 assert r1 1                 jmp r0</pre>
<pre>g :      malloc r2 1                 store r2 0                 move pc r3                 lea r3 ...                 crtcls [(x, r2)] r3                 jmp r0</pre>		

## The “awkward” example (naive translation)

<pre>g = fun _ =&gt;     let x = 0 in         fun f =&gt;             x := 0;             f();             x := 1;             f();             assert(x == 1)</pre> <hr/>	<pre>f : store x 0                 jmp r1 ! store x 1                 jmp r1 ! load r1 x assert r1 1 jmp r0</pre>
<pre>g : malloc r2 1     store r2 0 move pc r3 lea r3 ... crtcls [(x, r2)] r3 jmp r0</pre>	

# Stack and stack capability

- ▶ local rwlx-capability

## Stack and stack capability

- ▶ local rwlx-capability
- ▶ Stack capability always points to the top element of the stack

## Stack and stack capability

- ▶ local rwx-capability
- ▶ Stack capability always points to the top element of the stack
- ▶ Only place one can store local capabilities

## Stack and stack capability

- ▶ local rwx-capability
- ▶ Stack capability always points to the top element of the stack
- ▶ Only place one can store local capabilities
- ▶ When a stack is available, we assume it is in register  $r_{stk}$

# Macros (1)

`scall  $r(\bar{r}_{args}, \bar{r}_{priv})$`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to

# Macros (1)

`scall  $r(\bar{r}_{args}, \bar{r}_{priv})$`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:

# Macros (1)

`scall  $r(\bar{r}_{args}, \bar{r}_{priv})$`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability

# Macros (1)

`scall r( $\bar{r}_{args}$ ,  $\bar{r}_{priv}$ )`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability
  - ▶ Create protected return pointer

# Macros (1)

`scall  $r(\bar{r}_{args}, \bar{r}_{priv})$`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability
  - ▶ Create protected return pointer
  - ▶ Restrict stack capability to unused part

# Macros (1)

`scall r( $\bar{r}_{args}$ ,  $\bar{r}_{priv}$ )`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability
  - ▶ Create protected return pointer
  - ▶ Restrict stack capability to unused part
  - ▶ Clear the part of the stack we release control over
  - ▶ Clear unused registers

# Macros (1)

`scall  $r(\bar{r}_{args}, \bar{r}_{priv})$`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability
  - ▶ Create protected return pointer
  - ▶ Restrict stack capability to unused part
  - ▶ Clear the part of the stack we release control over
  - ▶ Clear unused registers
  - ▶ Jump to  $r$

# Macros (1)

`scall r( $\bar{r}_{args}$ ,  $\bar{r}_{priv}$ )`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability
  - ▶ Create protected return pointer
  - ▶ Restrict stack capability to unused part
  - ▶ Clear the part of the stack we release control over
  - ▶ Clear unused registers
  - ▶ Jump to  $r$
  - ▶ Upon return: Run the on stack restoration code

# Macros (1)

`scall  $r(\bar{r}_{args}, \bar{r}_{priv})$`

- ▶  $\bar{r}_{args}$  list of argument registers
- ▶  $\bar{r}_{priv}$  list of “private” registers
- ▶  $r$  register to jump to
- ▶ `scall` does the following:
  - ▶ Push
    - ▶ the restore code to the stack.
    - ▶ “private” registers to the stack.
    - ▶ return address capability
    - ▶ stack capability
  - ▶ Create protected return pointer
  - ▶ Restrict stack capability to unused part
  - ▶ Clear the part of the stack we release control over
  - ▶ Clear unused registers
  - ▶ Jump to  $r$
  - ▶ Upon return: Run the on stack restoration code
  - ▶ Return address in caller-code: Restore “private” state

## The “awkward” example (naive translation)

```
g = fun _ =>
    let x = 0 in
    fun f =>
        x := 0;
        f();
        x := 1;
        f();
        assert(x == 1)
```

---

```
g :   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      jmp r0
```

```
f :   store x 0
      jmp r1
      store x 1
      jmp r1
      load r1 x
      assert r1 1
      jmp r0
```

## The “awkward” example (naive translation)

```
g = fun _ =>
    let x = 0 in
    fun f =>
        x := 0;
        f();
        x := 1;
        f();
        assert(x == 1)
```

---

```
g :   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      jmp r0
```

```
f :   store x 0
      scall r1([], [r0, r1])
      store x 1
      scall r1([], [r0])
      load r1 x
      assert r1 1
      jmp r0
```

## The “awkward” example (naive translation)

```
g = fun _ =>
    let x = 0 in
    fun f =>
        x := 0;
        f();
        x := 1;
        f();
        assert(x == 1)
```

---

```
g :   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      jmp r0 !
```

```
f :   store x 0
      scall r1([], [r0, r1])
      store x 1
      scall r1([], [r0])
      load r1 x
      assert r1 1
      jmp r0 !
```

## Macros (2)

► `mclear r`

## Macros (2)

- ▶ `mclear r`
  - ▶ Clear all memory cells capability in register  $r$  has authority over.

## Macros (2)

- ▶ `mclear r`
  - ▶ Clear all memory cells capability in register *r* has authority over.
- ▶ `rclear r̄`

## Macros (2)

- ▶ `mclear r`
  - ▶ Clear all memory cells capability in register *r* has authority over.
- ▶ `rclear r̄`
  - ▶ Clear all the registers in *r̄*.

## The “awkward” example (naive translation)

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
    assert(x == 1)
```

```
f : store x 0
scall r1([], [r0, r1])
store x 1
```

---

```
g : malloc r2 1
store r2 0
move pc r3
lea r3 ...
crtcls [(x, r2)] r3
jmp r0
```

```
load r1 x
assert r1 1
```

## The “awkward” example (naive translation)

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
    assert(x == 1)
```

---

```
g :  malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      rclear RN \ {pc, r0, r1}
      jmp r0
```

```
f :  store x 0
      scall r1([], [r0, r1])
      store x 1
      scall r1([], [r0])
      load r1 x
      assert r1 1
```

```
rclear RN \ {r0, pc}
jmp r0
```

## The “awkward” example (naive translation)

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
    assert(x == 1)
```

---

```
g :  malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      rclear RN \ {pc, r0, r1}
      jmp r0
```

```
f :  store x 0
      scall r1([], [r0, r1])
      store x 1
      scall r1([], [r0])
      load r1 x
      assert r1 1
      mclear rstk
      rclear RN \ {r0, pc}
      jmp r0
```

## The “awkward” example (naive translation)

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
    assert(x == 1)
```

---

```
g :  malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      rclear RN \ {pc, r0, r1}
      jmp r0
```

```
f :  store x 0 !
      scall r1([], [r0, r1])
      store x 1
      scall r1([], [r0])
      load r1 x
      assert r1 1
      mclear rstk
      rclear RN \ {r0, pc}
      jmp r0
```

## Macros (3)

► reqglob r

## Macros (3)

- ▶ **reqglob *r***
  - ▶ if the word in register *r* is not a global capability, then fail

## Macros (3)

- ▶ **reqglob *r***
  - ▶ if the word in register *r* is not a global capability, then fail
  - ▶ otherwise continue execution

## Macros (3)

- ▶ `reqglob r`
  - ▶ if the word in register *r* is not a global capability, then fail
  - ▶ otherwise continue execution
- ▶ `prepstack r`

## Macros (3)

- ▶ `reqglob r`
  - ▶ if the word in register  $r$  is not a global capability, then fail
  - ▶ otherwise continue execution
- ▶ `prepstack r`
  - ▶ if the word in register  $r$  is not an rwlx-capability, then fail

## Macros (3)

- ▶ **reqglob *r***
  - ▶ if the word in register *r* is not a global capability, then fail
  - ▶ otherwise continue execution
- ▶ **prepstack *r***
  - ▶ if the word in register *r* is not an rwx-capability, then fail
  - ▶ otherwise set pointer of capability to point just below range of authority

## The “awkward” example (final version)

```
g = fun _ =>
  let x = 0 in
    fun f =>
      x := 0;
      f();
      x := 1;
      f();
      assert(x == 1)
```

---

```
g:   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      rclear RN \ {pc, r0, r1}
      jmp r0
```

f :

```
store x 0
scall r1([], [r0, r1])
store x 1
scall r1([], [r0])
load r1 x
assert r1 1
mclear rstk
rclear RN \ {r0, pc}
jmp r0
```

## The “awkward” example (final version)

```
g = fun _ =>
    let x = 0 in
    fun f =>
        x := 0;
        f();
        x := 1;
        f();
        assert(x == 1)
```

---

```
g:   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      rclear RN \ {pc, r0, r1}
      jmp r0
```

f :

```
prepstack rstk
store x 0
scall r1([], [r0, r1])
store x 1
scall r1([], [r0])
load r1 x
assert r1 1
mclear rstk
rclear RN \ {r0, pc}
jmp r0
```

## The “awkward” example (final version)

```
g = fun _ =>
  let x = 0 in
    fun f =>
      x := 0;
      f();
      x := 1;
      f();
      assert(x == 1)
```

---

```
g :   malloc r2 1
      store r2 0
      move pc r3
      lea r3 ...
      crtcls [(x, r2)] r3
      rclear RN \ {pc, r0, r1}
      jmp r0
```

```
f :   reqglob r1
      prepstack rstk
      store x 0
      scall r1([], [r0, r1])
      store x 1
      scall r1([], [r0])
      load r1 x
      assert r1 1
      mclear rstk
      rclear RN \ {r0, pc}
      jmp r0
```

## Stack discipline

- ▶ *Always* clear the unused stack before transferring control to untrusted code.

## Stack discipline

- ▶ *Always* clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.

## Stack discipline

- ▶ *Always* clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.
  - ▶ Prevent adversary from “hiding” local capability on the stack

## Stack discipline

- ▶ Always clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.
  - ▶ Prevent adversary from “hiding” local capability on the stack
- ▶ If stack capability untrusted, then

# Stack discipline

- ▶ Always clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.
  - ▶ Prevent adversary from “hiding” local capability on the stack
- ▶ If stack capability untrusted, then
  - ▶ only use it if it is rwx

## Stack discipline

- ▶ Always clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.
  - ▶ Prevent adversary from “hiding” local capability on the stack
- ▶ If stack capability untrusted, then
  - ▶ only use it if it is rwx
  - ▶ make it point just below range of authority.

## Stack discipline

- ▶ Always clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.
  - ▶ Prevent adversary from “hiding” local capability on the stack
- ▶ If stack capability untrusted, then
  - ▶ only use it if it is rwx
  - ▶ make it point just below range of authority.
  - ▶ If it looks like a stack, works like a stack, and quacks like a stack, then it is probably a stack.

## Stack discipline

- ▶ Always clear the unused stack before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities on the stack.
  - ▶ Prevent adversary from “hiding” local capability on the stack
- ▶ If stack capability untrusted, then
  - ▶ only use it if it is rwx
  - ▶ make it point just below range of authority.
  - ▶ If it looks like a stack, works like a stack, and quacks like a stack, then it is probably a stack.
- ▶ In the presence of an untrusted stack capability, only use global callbacks.

## Register discipline

- ▶ *Always clear non-argument registers before transferring control to untrusted code.*

## Register discipline

- ▶ Always clear non-argument registers before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities.

## Register discipline

- ▶ *Always* clear non-argument registers before transferring control to untrusted code.
  - ▶ Prevent unintentionally leaking capabilities.
  - ▶ Prevent adversary from “hiding” local capability in a register.

# Bruges



# Road map

Capability Machine

Formalisation

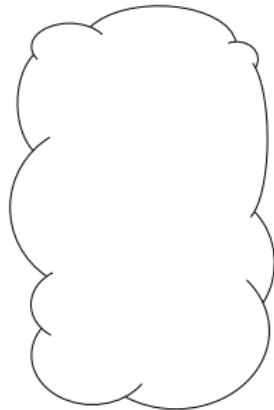
Example, Macros, and Stack Discipline

Logical Relation

Conclusion

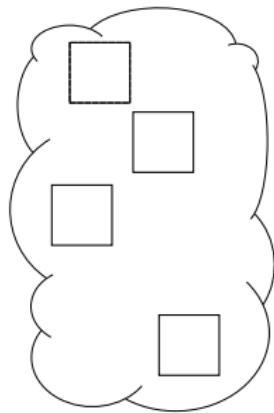
# Worlds

World



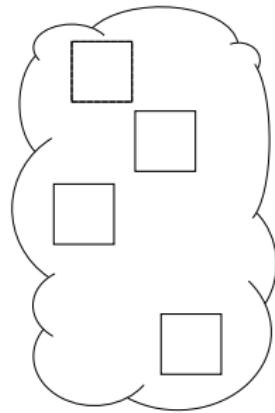
# Worlds

## World

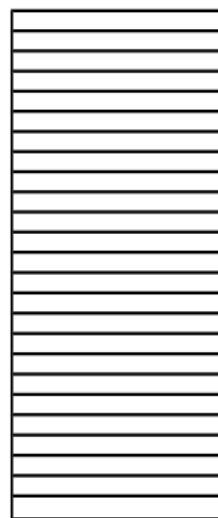


# Worlds

World



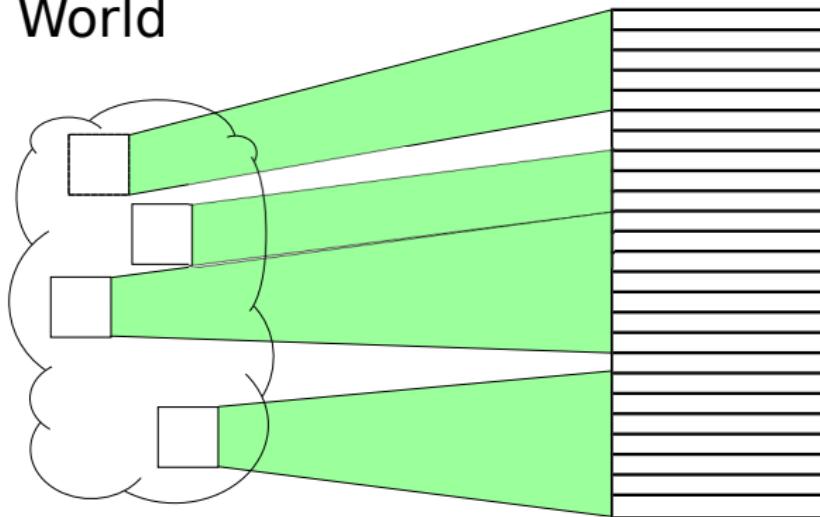
Memory



# Worlds

World

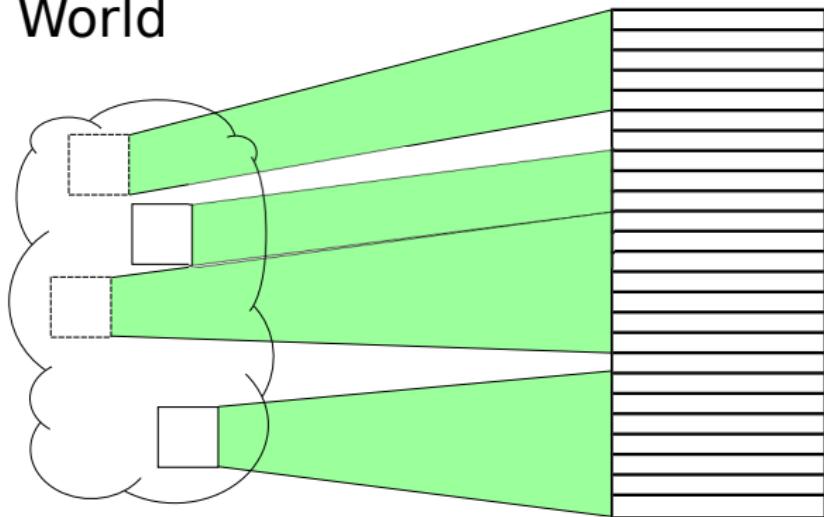
Memory



# Worlds

World

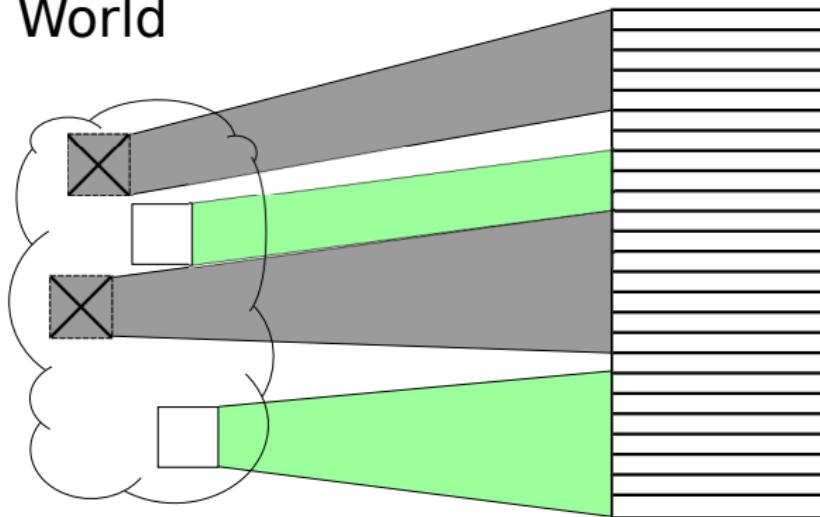
Memory



# Worlds

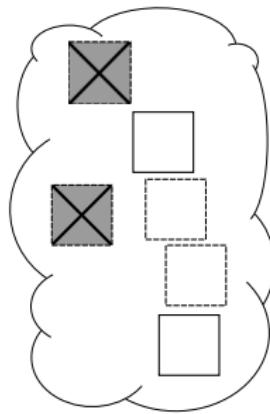
World

Memory

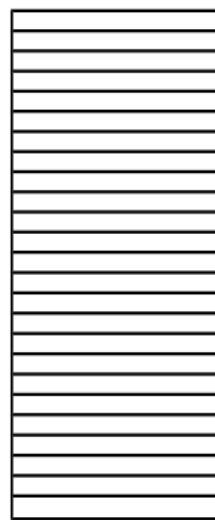


# Worlds

World

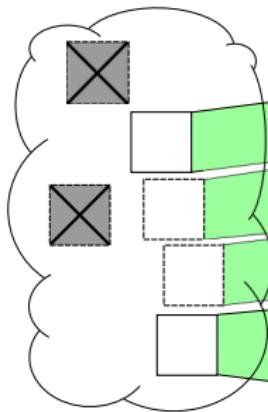


Memory



# Worlds

World



Memory

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\text{fin}} \text{Region}$

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\textit{fin}} \text{Region}$
- ▶ Three kinds of regions:

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\text{fin}}$  Region
- ▶ Three kinds of regions:
  - permanent Models parts of memory that global and local capabilities can govern.

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\textit{fin}}$  Region
- ▶ Three kinds of regions:
  - permanent** Models parts of memory that global and local capabilities can govern.
  - temporary** Models parts of memory that only local capabilities can govern.

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\text{fin}}$  Region
- ▶ Three kinds of regions:
  - permanent** Models parts of memory that global and local capabilities can govern.
  - temporary** Models parts of memory that only local capabilities can govern.
  - revoked** Masking of region.

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\text{fin}}$  Region
- ▶ Three kinds of regions:
  - permanent** Models parts of memory that global and local capabilities can govern.
  - temporary** Models parts of memory that only local capabilities can govern.
  - revoked** Masking of region.
- ▶ Two future world relations

# Worlds

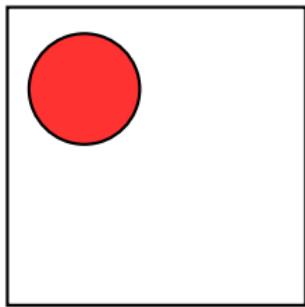
- ▶ World :  $\mathbb{N} \xrightarrow{\textit{fin}}$  Region
- ▶ Three kinds of regions:
  - permanent** Models parts of memory that global and local capabilities can govern.
  - temporary** Models parts of memory that only local capabilities can govern.
  - revoked** Masking of region.
- ▶ Two future world relations
  - public** Extensional ( $\sqsupseteq^{\textit{pub}}$ )

# Worlds

- ▶ World :  $\mathbb{N} \xrightarrow{\textit{fin}}$  Region
- ▶ Three kinds of regions:
  - permanent** Models parts of memory that global and local capabilities can govern.
  - temporary** Models parts of memory that only local capabilities can govern.
  - revoked** Masking of region.
- ▶ Two future world relations
  - public** Extensional ( $\sqsupseteq^{\textit{pub}}$ )
  - private** Extensional and temporary regions can be revoked! ( $\sqsupseteq^{\textit{priv}}$ )

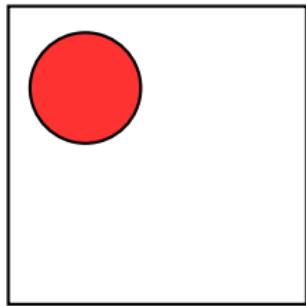
# Regions

## Region



# Regions

Region

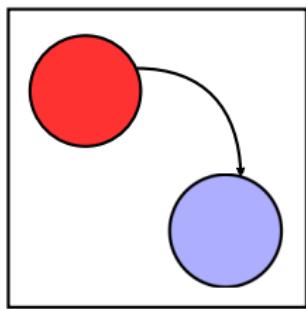


Memory cell



# Regions

Region

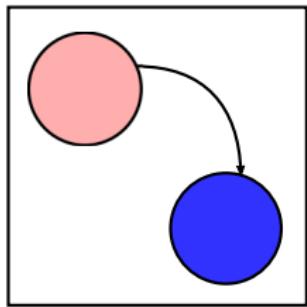


Memory cell



# Regions

Region

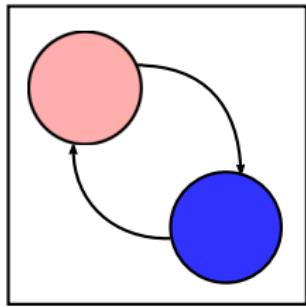


Memory cell



# Regions

Region

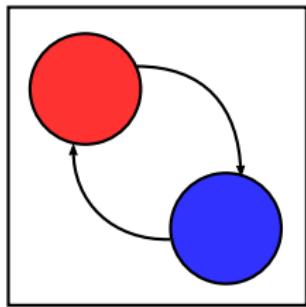


Memory cell



# Regions

Region

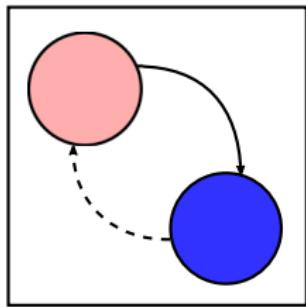


Memory cell



# Regions

Region



Memory cell



# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive
  - ▶  $H$  : State  $\rightarrow$  World  $\xrightarrow{mon}$  Pred(MemSegment), state interpretation function

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive
  - ▶  $H$  : State → World  $\xrightarrow{mon}$  Pred(MemSegment), state interpretation function
    - ▶ if region permanent, then  $H$  monotone w.r.t.  $\sqsupseteq^{priv}$

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive
  - ▶  $H$  : State  $\rightarrow$  World  $\xrightarrow{mon}$  Pred(MemSegment), state interpretation function
    - ▶ if region permanent, then  $H$  monotone w.r.t.  $\sqsupseteq^{priv}$
    - ▶ if region temporary, then  $H$  monotone w.r.t.  $\sqsupseteq^{pub}$

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive
  - ▶  $H$  : State → World  $\xrightarrow{mon}$  Pred(MemSegment), state interpretation function
    - ▶ if region permanent, then  $H$  monotone w.r.t.  $\sqsupseteq^{priv}$
    - ▶ if region temporary, then  $H$  monotone w.r.t.  $\sqsupseteq^{pub}$
- ▶ Future Worlds

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive
  - ▶  $H$  : State → World  $\xrightarrow{mon}$  Pred(MemSegment), state interpretation function
    - ▶ if region permanent, then  $H$  monotone w.r.t.  $\sqsupseteq^{priv}$
    - ▶ if region temporary, then  $H$  monotone w.r.t.  $\sqsupseteq^{pub}$
- ▶ Future Worlds
  - ▶ In  $\sqsupseteq^{pub}$ , existing regions are allowed to make transitions in  $\phi_{pub}$

# Regions

- ▶ Regions are transition systems  $(v, s, \phi_{pub}, \phi, H)$ 
  - ▶  $v$ , view
  - ▶  $s$ , current state
  - ▶  $\phi_{pub}$  : State<sup>2</sup>, reflexive and transitive
  - ▶  $\phi \supseteq \phi_{pub}$ , reflexive and transitive
  - ▶  $H$  : State → World  $\xrightarrow{mon}$  Pred(MemSegment), state interpretation function
    - ▶ if region permanent, then  $H$  monotone w.r.t.  $\sqsubseteq^{priv}$
    - ▶ if region temporary, then  $H$  monotone w.r.t.  $\sqsubseteq^{pub}$
- ▶ Future Worlds
  - ▶ In  $\sqsubseteq^{pub}$ , existing regions are allowed to make transitions in  $\phi_{pub}$
  - ▶ In  $\sqsubseteq^{priv}$ , existing regions are allowed to make transitions in  $\phi$

# Logical Relation

- ▶ Step indexed

# Logical Relation

- ▶ Step indexed
  - ▶ but in the following, the steps are omitted.

## Observation relation

- ▶ Executable configurations that produce desired results.

$\mathcal{O} : \text{World} \rightarrow \text{Pred}(\text{Reg} \times \text{MemSegment})$

$$\mathcal{O}(W) \stackrel{\text{def}}{=} \{(reg, ms) \mid$$

## Observation relation

- ▶ Executable configurations that produce desired results.

$$\mathcal{O} : \text{World} \rightarrow \text{Pred}(\text{Reg} \times \text{MemSegment})$$

$$\begin{aligned}\mathcal{O}(W) \stackrel{\text{def}}{=} & \{(reg, ms) \mid \forall ms_f, mem'. \\ & (reg, ms \uplus ms_f) \rightarrow (halted, mem')\end{aligned}$$

## Observation relation

- ▶ Executable configurations that produce desired results.

$\mathcal{O} : \text{World} \rightarrow \text{Pred}(\text{Reg} \times \text{MemSegment})$

$$\begin{aligned}\mathcal{O}(W) \stackrel{\text{def}}{=} & \{(reg, ms) \mid \forall ms_f, mem'. \\ & (reg, ms \uplus ms_f) \rightarrow (halted, mem') \\ & \Rightarrow \exists W' \sqsupseteq^{\text{priv}} W. \exists ms_r, ms'. \\ & mem' = ms' \uplus ms_r \uplus ms_f \wedge \\ & ms' : W'\}\end{aligned}$$

## Register-file relation

- ▶ Register-files with “well-behaved” words.
- ▶ On jump, the contents of the register-file can be seen as the arguments.

$$\begin{aligned}\mathcal{R} : \text{World} &\xrightarrow{\stackrel{\textit{mon}}{\sqsupseteq^{\textit{pub}}}} \text{Pred}(\text{Reg}) \\ \mathcal{R}(W) &\stackrel{\textit{def}}{=}\end{aligned}$$

## Register-file relation

- ▶ Register-files with “well-behaved” words.
- ▶ On jump, the contents of the register-file can be seen as the arguments.

$$\begin{aligned}\mathcal{R} : \text{World} &\xrightarrow[\sqsupseteq^{\text{pub}}]{\text{\scriptsize mon}} \text{Pred(Reg)} \\ \mathcal{R}(W) &\stackrel{\text{\scriptsize def}}{=} \{ \textit{reg} \mid \forall r \in \text{RegisterName} \setminus \{\text{pc}\}. \\ &\quad \textit{reg}(r) \in \mathcal{V}(W) \}\end{aligned}$$

# Expression relation

- ▶ Words that produce admissible results when “executed”.

$\mathcal{E} : \text{World} \rightarrow \text{Pred}(\text{Word})$

$$\mathcal{E}(W) \stackrel{\text{def}}{=}$$

# Expression relation

- ▶ Words that produce admissible results when “executed”.

$\mathcal{E} : \text{World} \rightarrow \text{Pred}(\text{Word})$

$\mathcal{E}(W) \stackrel{\text{def}}{=} \{c \mid \forall reg \in \mathcal{R}(W).$

$\forall ms : W.$

# Expression relation

- ▶ Words that produce admissible results when “executed”.

$\mathcal{E} : \text{World} \rightarrow \text{Pred}(\text{Word})$

$$\mathcal{E}(W) \stackrel{\text{def}}{=} \{c \mid \forall reg \in \mathcal{R}(W).$$

$\forall ms : W.$

$$(reg[\text{pc} \mapsto pc], ms) \in \mathcal{O}(W)\}$$

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants
- ▶ Global capabilities

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants
- ▶ Global capabilities
  - ▶ Must respect  $\sqsupseteq^{priv}$

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants
- ▶ Global capabilities
  - ▶ Must respect  $\sqsupseteq^{priv}$
  - ▶ Authority only over memory segments governed by permanent region.

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants
- ▶ Global capabilities
  - ▶ Must respect  $\sqsupseteq^{priv}$
  - ▶ Authority only over memory segments governed by permanent region.
- ▶ Local capabilities

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants
- ▶ Global capabilities
  - ▶ Must respect  $\sqsupseteq^{priv}$
  - ▶ Authority only over memory segments governed by permanent region.
- ▶ Local capabilities
  - ▶ Must respect  $\sqsupseteq^{pub}$

# Value relation

## $\mathcal{V}$ -relation

- ▶ All integers (data) are in the  $\mathcal{V}$ -relation
- ▶ For capabilities, define a condition for each kind of permission it grants
- ▶ Global capabilities
  - ▶ Must respect  $\sqsupseteq^{priv}$
  - ▶ Authority only over memory segments governed by permanent region.
- ▶ Local capabilities
  - ▶ Must respect  $\sqsupseteq^{pub}$
  - ▶ Authority over memory segments governed by either permanent or temporary regions

## Read condition

$$\begin{aligned} \text{readCondition}(g, W, \text{base}, \text{end}) = \\ \{(base, end) \mid \exists r \in \text{localityReg}(g, W). \\ \quad \exists [base', end'] \supseteq [base, end]. \\ \quad W(r) \subset \iota_{base', end'}^{\text{pwl}}\} \end{aligned}$$

## Read condition

$$\begin{aligned} \text{readCondition}(g, W, \text{base}, \text{end}) = \\ \{(base, end) \mid \exists r \in \text{localityReg}(g, W). \\ \quad \exists [base', end'] \supseteq [base, end]. \\ \quad W(r) \subset \iota_{base', end'}^{\text{pwl}}\} \\ \iota_{base, end}^{\text{pwl}} \stackrel{\text{def}}{=} (\text{temp}, 1, =, =, H_{base, end}^{\text{pwl}}) \end{aligned}$$

## Read condition

$$\begin{aligned} \text{readCondition}(g, W, \text{base}, \text{end}) = \\ \{(base, end) \mid \exists r \in \text{localityReg}(g, W). \\ \exists [base', end'] \supseteq [base, end]. \\ W(r) \subset \iota_{base', end'}^{\text{pwl}}\} \end{aligned}$$

$$\iota_{base, end}^{\text{pwl}} \stackrel{\text{def}}{=} (\text{temp}, 1, =, =, H_{base, end}^{\text{pwl}})$$

$$H^{\text{pwl}} : \text{Addr}^2 \rightarrow \text{State} \rightarrow (\text{Wor}_{\sqsubseteq^{\text{pub}}} \xrightarrow{\text{mon, ne}} \text{Pred}(\text{MemSegment}))$$

$$H_{base, end}^{\text{pwl}} \circ \hat{W} \stackrel{\text{def}}{=} \left\{ ms \middle| \begin{array}{l} \text{dom}(ms) = [base, end] \wedge \\ \forall a \in [base, end]. ms(a) \in \mathcal{V}(\hat{W}) \end{array} \right\}$$

## Write condition

$$\begin{aligned} \text{writeCondition}(\iota, g, W, \text{base}, \text{end}) = \\ \{ (\text{base}, \text{end}) \mid \exists r \in \text{localityReg}(g, W). \\ \quad \exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \\ \quad W(r) \supset \iota_{\text{base}', \text{end}'} \} \end{aligned}$$

## Write condition

$$\begin{aligned} \text{writeCondition}(\iota, g, W, \text{base}, \text{end}) = \\ \{(base, end) \mid \exists r \in \text{localityReg}(g, W). \\ \quad \exists [base', end'] \supseteq [base, end]. \\ \quad W(r) \supset \iota_{base', end'}\} \end{aligned}$$

$$\iota_{base, end}^{nwl} \stackrel{\text{def}}{=} (\text{temp}, 1, =, =, H_{start, end}^{nwl})$$

## Write condition

$$\begin{aligned} \text{writeCondition}(\iota, g, W, \text{base}, \text{end}) = \\ \{(base, end) \mid \exists r \in \text{localityReg}(g, W). \\ \exists [base', end'] \supseteq [base, end]. \\ W(r) \supset \iota_{base', end'}\} \end{aligned}$$

$$\iota_{base, end}^{nwl} \stackrel{\text{def}}{=} (\text{temp}, 1, =, =, H_{start, end}^{nwl})$$

$$H^{nwl} : \text{Addr}^2 \rightarrow \text{State} \rightarrow (\text{Wor}_{\sqsupseteq^{\text{priv}}} \xrightarrow{\text{mon, ne}} \text{Pred}(\text{MemSegment}))$$

$$H_{base, end}^{nwl} \ s \ \hat{W} \stackrel{\text{def}}{=} \left\{ ms \middle| \begin{array}{l} \text{dom}(ms) = [base, end] \wedge \\ \forall a \in [base, end]. \\ ms(a) \in \mathcal{V}(\text{revokeTemp}(\hat{W})) \end{array} \right\}$$

## Conditions on execution

$$\begin{aligned} \text{executeCondition}(g, W, perm, base, end) = \\ \{(perm, base, end) \mid &\forall W' \sqsupseteq W. \\ &\forall a \in [base, end]. \\ &((perm, g), base, end, a) \in \mathcal{E}(W')\} \end{aligned}$$

## Conditions on execution

$$\begin{aligned} \text{executeCondition}(g, W, perm, base, end) = \\ \{(perm, base, end) \mid & \forall W' \sqsupseteq W. \\ & \forall a \in [base, end]. \\ & ((perm, g), base, end, a) \in \mathcal{E}(W')\} \end{aligned}$$

where  $g = \text{local} \Rightarrow \sqsupseteq = \sqsupseteq^{\text{pub}}$

and  $g = \text{global} \Rightarrow \sqsupseteq = \sqsupseteq^{\text{priv}}$

## Conditions on execution

$\text{executeCondition}(g, W, perm, base, end) =$

$\{(perm, base, end) \mid \forall W' \sqsupseteq W.$

$\forall a \in [base, end].$

$((perm, g), base, end, a) \in \mathcal{E}(W')\}$

$\text{enterCondition}(g, W, base, end, a) =$

$\{(base, end, a) \mid \forall W' \sqsupseteq W.$

$((rx, g), base, end, a) \in \mathcal{E}(W')\}$

where  $g = \text{local} \Rightarrow \sqsupseteq = \sqsupseteq^{\text{pub}}$

and  $g = \text{global} \Rightarrow \sqsupseteq = \sqsupseteq^{\text{priv}}$

# Value relation

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{\textit{pub}}]{\textit{mon}} \text{Pred}(\text{Word})$$

$$\mathcal{V}(W) \stackrel{\textit{def}}{=}$$

## Value relation

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{\textit{pub}}]{\textit{mon}} \text{Pred}(\text{Word})$$

$$\mathcal{V}(W) \stackrel{\textit{def}}{=} \{i \mid i \in \mathbb{Z}\} \cup$$

## Value relation

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{\text{pub}}]{\text{mon}} \text{Pred}(\text{Word})$$

$$\begin{aligned}\mathcal{V}(W) \stackrel{\text{def}}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\ & \{((o, g), \text{base}, \text{end}, a)\} \cup\end{aligned}$$

## Value relation

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{\text{pub}}]{\text{mon}} \text{Pred}(\text{Word})$$

$$\begin{aligned}\mathcal{V}(W) \stackrel{\text{def}}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\ & \{((o, g), \text{base}, \text{end}, a)\} \cup \\ & \{((e, g), \text{base}, \text{end}, a) \mid \\ & \quad \text{enterCondition}(g, W, \text{base}, \text{end}, a)\} \cup\end{aligned}$$

## Value relation

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{\text{pub}}]{\text{mon}} \text{Pred}(\text{Word})$$

$$\begin{aligned}\mathcal{V}(W) \stackrel{\text{def}}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\ & \{((o, g), \text{base}, \text{end}, a)\} \cup \\ & \{((e, g), \text{base}, \text{end}, a) \mid \\ & \quad \text{enterCondition}(g, W, \text{base}, \text{end}, a)\} \cup \\ & \{((\text{rwlx}, g), \text{base}, \text{end}, a) \mid \\ & \quad \text{readCondition}(g, W, \text{base}, \text{end}) \wedge \\ & \quad \text{writeCondition}(\iota^{\text{PWL}}, g, W, \text{base}, \text{end}) \wedge \\ & \quad \text{executeCondition}(g, W, \text{rwlx}, \text{base}, \text{end})\} \cup\end{aligned}$$

## Value relation

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{\text{pub}}]{\text{mon}} \text{Pred}(\text{Word})$$

$$\begin{aligned}\mathcal{V}(W) \stackrel{\text{def}}{=} & \{i \mid i \in \mathbb{Z}\} \cup \\ & \{((o, g), \text{base}, \text{end}, a)\} \cup \\ & \{((e, g), \text{base}, \text{end}, a) \mid \\ & \quad \text{enterCondition}(g, W, \text{base}, \text{end}, a)\} \cup \\ & \{((\text{rwlx}, g), \text{base}, \text{end}, a) \mid \\ & \quad \text{readCondition}(g, W, \text{base}, \text{end}) \wedge \\ & \quad \text{writeCondition}(\iota^{\text{PWL}}, g, W, \text{base}, \text{end}) \wedge \\ & \quad \text{executeCondition}(g, W, \text{rwlx}, \text{base}, \text{end})\} \cup \\ & \dots\end{aligned}$$

# Fundamental Theorem of Logical Relations (FTLR)

Lemma (FTLR)

*For all  $W \in \text{World}$  and  $c \in \text{Caps}$ ,*

$$c \in \mathcal{E}(W).$$

# Fundamental Theorem of Logical Relations (FTLR)

## Lemma (FTLR)

*For all  $W \in \text{World}$  and  $c \in \text{Caps}$ ,*

$$c \in \mathcal{E}(W).$$

- ▶ The pc-register can be accessed like any other register

# Fundamental Theorem of Logical Relations (FTLR)

## Lemma (FTLR)

*For all  $W \in \text{World}$  and  $c \in \text{Caps}$ ,*

$$c \in \mathcal{E}(W).$$

- ▶ The pc-register can be accessed like any other register
- ▶ Capability must behave when used for read/write

# Fundamental Theorem of Logical Relations (FTLR)

## Lemma (FTLR)

For all  $W \in \text{World}$ ,  $g \in \text{Global}$ ,  $\text{perm} \in \text{Perm}$ , and  
 $\text{base}, \text{end}, a \in \text{Addr}$ ,

if

$\text{perm} = \text{rx}$  and  $\text{readCondition}(g, W, \text{base}, \text{end})$ ,

or

$\text{perm} = \text{rwx}$  and  $\text{readCondition}(W, \text{base}, \text{end})$

and  $\text{writeCondition}(\iota^{nwl}, g, W, \text{base}, \text{end})$

or

...

then

$(\text{perm}, \text{base}, \text{end}, a) \in \mathcal{E}(W)$ .

## The awkward example

- ▶ Using the logical relation, we can prove well-bracketedness for the awkward example.

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
  assert(x == 1)
```

## The awkward example

- ▶ Using the logical relation, we can prove well-bracketedness for the awkward example.
- ▶ The proof will have to wait for another time.

```
g = fun _ =>
  let x = 0 in
  fun f =>
    x := 0;
    f();
    x := 1;
    f();
  assert(x == 1)
```

# Namur



# Road map

Capability Machine

Formalisation

Example, Macros, and Stack Discipline

Logical Relation

Conclusion

# Conclusion

- ▶ With a simple capability system and reasonable conventions, we can enforce well-bracketedness.
- ▶ Using known logical relation techniques, we can reason about programs for a simple capability machine.

# Questions?

# Questions?

- ▶ Chocolates in the kitchen.

## References

- [1] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *International Symposium on Computer Architecture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.