Reasoning about capability machines using logical relations

Lau Skorstengaard

Aarhus University

KU Leuven, November 2016

Road map

Capability Machine

Formalisation

Example program

Logical Relation

Example revisited

Current work

Road map

- Capability Machine
- Formalisation
- Example program
- Logical Relation
- Example revisited
- Current work

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

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

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

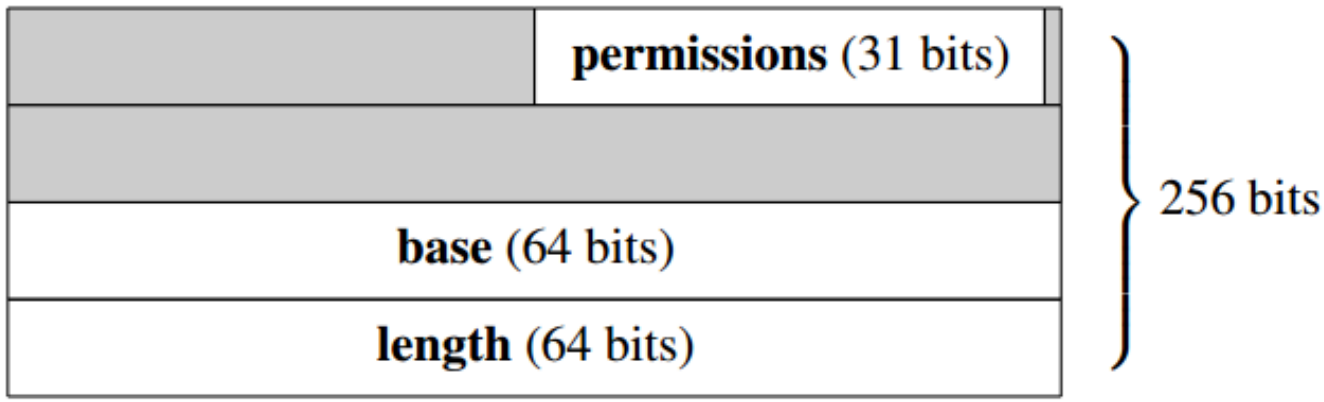


Figure: CHERI capability [1]

Capabilities

What is a capability?

- *Unforgeable* token of authority

What is a capability in a capability machine?

- Unforgeable pointer
- Range of memory

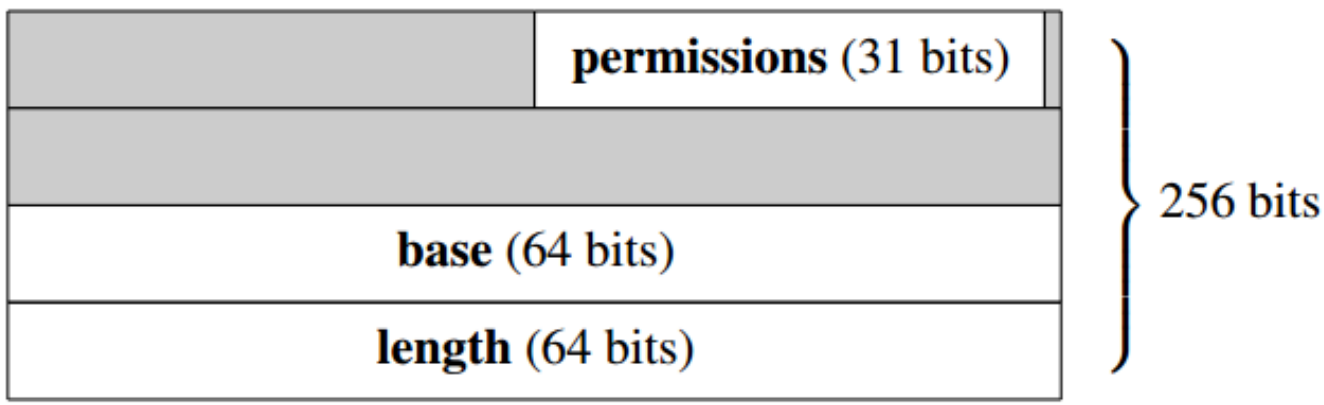


Figure: CHERI capability [1]

Capabilities

What is a capability?

- *Unforgeable* token of authority

What is a capability in a capability machine?

- Unforgeable pointer
- Range of memory
- Permission

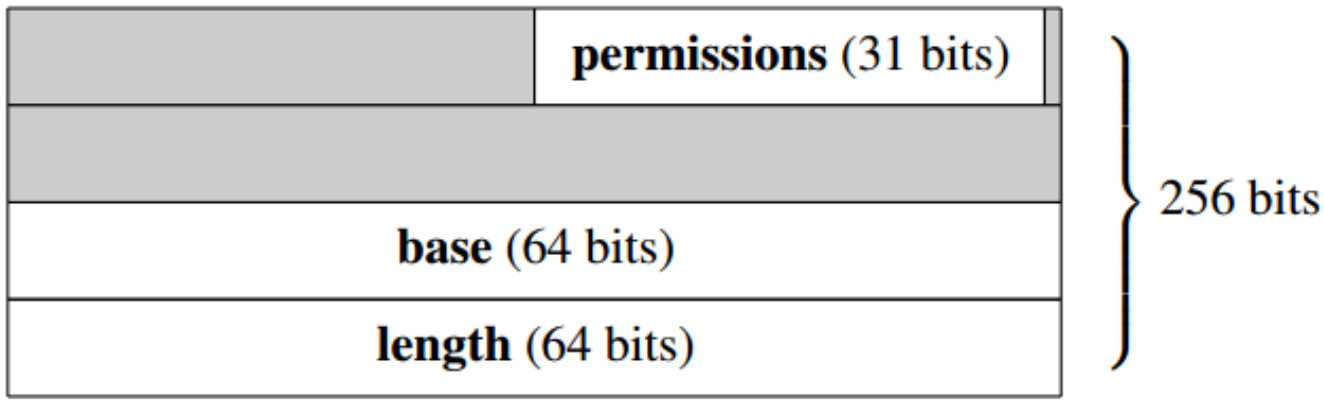


Figure: CHERI capability [1]

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 domains

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 domains
  - 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
- Heap and registers

Capability machine overview

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

Road map

Capability Machine

Formalisation

Example program

Logical Relation

Example revisited

Current work

Formalisation

- A mathematical model of the system

Formalisation

- A mathematical model of the system
- Allows us to reason formally

Formalisation

- A mathematical model of the system
- Allows us to reason formally
- May make some abstractions

Formalisation

- A mathematical model of the system
- Allows us to reason formally
- May make some abstractions
- Needs to stay true to a real system

Formalisation

- A mathematical model of the system
- Allows us to reason formally
- May make some abstractions
- Needs to stay true to a real system
- This formalisation is of a capability machine (not CHERI or the M-Machine)

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{ \qquad \}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o},\qquad\qquad\qquad\qquad\}$$

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \qquad\qquad\qquad\qquad \}$$

Formalisation - Permissions

Permissions

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

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

Formalisation - Permissions

Permissions

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

$$\mathrm{Perm} \stackrel{def}{=} \{\mathrm{o}, \mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \qquad\}$$

Formalisation - Permissions

Permissions

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

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

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

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

Formalisation - Capabilities

Capability

$$\mathrm{Cap} \stackrel{def}{=}$$

Formalisation - Capabilities

Capability

- Permission

$$\mathrm{Cap} \stackrel{def}{=}$$

Formalisation - Capabilities

Capability

- Permission

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm}$$

Formalisation - Capabilities

Capability

- Permission
- Range of authority

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm}$$

Formalisation - Capabilities

Capability

- Permission
- Range of authority

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm}$$

Formalisation - Capabilities

Capability

- Permission
- Range of authority

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm} \times \mathrm{Addr} \times \mathrm{Addr}$$

Formalisation - Capabilities

Capability

- Permission
- Range of authority
- Pointer

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm} \times \mathrm{Addr} \times \mathrm{Addr}$$

Formalisation - Capabilities

Capability

- Permission
- Range of authority
- Pointer

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm} \times \mathrm{Addr} \times \mathrm{Addr} \times \mathrm{Addr}$$

Formalisation - Capabilities

Capability

- Permission
- Range of authority
- Pointer

$$\mathrm{Addr} \stackrel{def}{=} \mathbb{N}$$

$$\mathrm{Cap} \stackrel{def}{=} \mathrm{Perm} \times \mathrm{Addr} \times \mathrm{Addr} \times \mathrm{Addr}$$

Example: $(\mathrm{e}, 30, 42, 30)$

Formalisation - Words and register file

Words

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

Formalisation - Words and register file

Words

- Capability

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

Formalisation - Words and register file

Words

- Capability

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

Formalisation - Words and register file

Words

- Capability
- Data (and instructions)

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

Formalisation - Words and register file

Words

- Capability
- Data (and instructions)

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

Formalisation - Words and register file

Words

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

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

Formalisation - Words and register file

Words

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

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

Register file

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

Formalisation - Words and register file

Words

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

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

Register file

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

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

Formalisation - Words and register file

Words

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

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

Register file

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

$$\mathrm{Reg} \stackrel{def}{=} \mathrm{RegisterName} \to \mathrm{Word}$$

Formalisation - Heap and configurations

Heap

$$\mathrm{Heap} \stackrel{def}{=}$$

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

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

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration

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

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Heap}$$

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration
- Successfully halted configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Heap}$$

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration
- Successfully halted configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Heap} \qquad + \{\mathit{halted}\} \times \mathrm{Heap}$$

Formalisation - Heap and configurations

Heap

- Map from Addr to Word

$$\mathrm{Heap} \stackrel{def}{=} \mathrm{Addr} \to \mathrm{Word}$$

Configuration

- Executable configuration
- Successfully halted configuration
- Failed configuration

$$\mathrm{Conf} \stackrel{def}{=} \mathrm{Reg} \times \mathrm{Heap} + \{\mathit{failed}\} + \{\mathit{halted}\} \times \mathrm{Heap}$$

Formalisation - Instructions

Syntax

$$\text{Instructions} \quad ::=$$

Formalisation - Instructions

Syntax

$$
\begin{array}{rcl}
rn & ::= & n \mid r \\
\text{Instructions} & ::= &
\end{array}
$$

Formalisation - Instructions

Syntax

- The normal instructions

$$\begin{aligned} rn &::= n \mid r \\ \text{Instructions} &::= \end{aligned}$$

Formalisation - Instructions

Syntax

- The normal instructions

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

Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

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

Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

$$
\begin{array}{rcl}
rn & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rn \mid \texttt{move}\ r\ rn \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rn\ rn \mid \\
 & & \texttt{lea}\ r\ rn \mid \texttt{restrict}\ r\ r\ rn \mid \\
 & & \texttt{subseg}\ r\ rn\ rn
\end{array}
$$

Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions
- Instructions for stopping the machine

$$
\begin{array}{rcl}
rn & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rn \mid \texttt{move}\ r\ rn \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rn\ rn \mid \\
 & & \texttt{lea}\ r\ rn \mid \texttt{restrict}\ r\ r\ rn \mid \\
 & & \texttt{subseg}\ r\ rn\ rn
\end{array}
$$

Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions
- Instructions for stopping the machine

$$
\begin{array}{rcl}
rn & ::= & n \mid r \\
\text{Instructions} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rn \mid \texttt{move}\ r\ rn \mid \\
 & & \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rn\ rn \mid \\
 & & \texttt{lea}\ r\ rn \mid \texttt{restrict}\ r\ r\ rn \mid \\
 & & \texttt{subseg}\ r\ rn\ rn \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, \mathit{a})}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, \mathit{a}) \qquad \mathit{base} \leq \mathit{a} \leq \mathit{end}}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, \mathit{a}) \qquad \mathit{base} \leq \mathit{a} \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}}{\mathit{executionAllowed}(\Phi)}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, \mathit{a}) \quad \mathit{base} \leq \mathit{a} \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}}{\mathit{executionAllowed}(\Phi)} \qquad \frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, \mathit{a}) \quad \mathit{base} \leq \mathit{a} \leq \mathit{end} \quad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}}{\mathit{executionAllowed}(\Phi)} \qquad \frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, \mathit{a}) \quad \mathit{base} \leq \mathit{a} \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}}{\mathit{executionAllowed}(\Phi)} \qquad \frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

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

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}}{\mathit{executionAllowed}(\Phi)}$$

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

$$\frac{\mathit{executionAllowed}(\Phi) \qquad i = \Phi.\mathrm{heap}(a)}{\Phi \rightarrow \ i}$$

Formalisation - Operational Semantics (1)

Execution relation

$$\rightarrow \subseteq (\mathrm{Reg} \times \mathrm{Heap}) \times \mathrm{Conf}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{base} \leq a \leq \mathit{end} \qquad \mathit{perm} \in \{\mathrm{rx}, \mathrm{rwx}\}}{\mathit{executionAllowed}(\Phi)} \qquad \frac{\neg \mathit{executionAllowed}(\Phi)}{\Phi \rightarrow \mathit{failed}}$$

$$\frac{\mathit{executionAllowed}(\Phi) \qquad i = \Phi.\mathrm{heap}(a)}{\Phi \rightarrow [\![i]\!](\Phi)}$$

Formalisation - Operational Semantics (2)

$$\llbracket \texttt{load}\ r_1\ r_2 \rrbracket (\Phi) =$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(r_2)}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = \Phi[\mathrm{reg}.r_1 \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a)}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\}}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = \Phi[\mathrm{reg}.r_1 \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{\begin{array}{cc} w = \Phi.\mathrm{heap}(a) & \Phi.\mathrm{reg}(r_2) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \\ \mathit{perm} \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} & \mathit{base} \leq a \leq \mathit{end} \end{array}}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = \qquad \Phi[\mathrm{reg}.r_1 \mapsto w]}$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (\mathit{perm}, \mathit{base}, \mathit{end}, a) \qquad \mathit{perm} \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad \mathit{base} \leq a \leq \mathit{end}}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{}{updatePc(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \qquad\qquad]}$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{\Phi.\mathrm{reg}(\mathrm{pc}) = (perm, base, end, a)}{updatePc(\Phi) = \Phi[\mathrm{reg}.\mathrm{pc} \mapsto \qquad\qquad]}$$

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\Phi.\mathrm{reg}(r_2) = (perm, base, end, a)$$

$$\frac{}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{\Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad newPerm = decodePerm(\Phi, r_3)}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad\qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{\Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad newPerm = decodePerm(\Phi, r_3) \qquad newPerm \sqsubseteq perm}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{\Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad newPerm = decodePerm(\Phi, r_3) \qquad newPerm \sqsubseteq perm \qquad c = (newPerm, base, end, a)}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = \qquad \Phi[\mathrm{reg}.r_1 \mapsto c]}$$

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

Formalisation - Operational Semantics (2)

$$\frac{w = \Phi.\mathrm{heap}(a) \qquad \Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad perm \in \{\mathrm{ro}, \mathrm{rw}, \mathrm{rx}, \mathrm{rwx}\} \qquad base \leq a \leq end}{[\![\texttt{load}\ r_1\ r_2]\!](\Phi) = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto w])}$$

$$\frac{\Phi.\mathrm{reg}(r_2) = (perm, base, end, a) \qquad newPerm = decodePerm(\Phi, r_3) \qquad newPerm \sqsubseteq perm \qquad c = (newPerm, base, end, a)}{[\![\texttt{restrict}\ r_1\ r_2\ r_3]\!] = updatePc(\Phi[\mathrm{reg}.r_1 \mapsto c])}$$

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

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 defined in a similar fashion

Road map

Capability Machine

Formalisation

Example program

Logical Relation

Example revisited

Current work

Example program

- High-level programs - ML style

Example program

- High-level programs - ML style
- `let l = 1 in ...` - allocates a new cell on the heap and sets the value to 1 (assume some trusted malloc exists).

Example program

- High-level programs - ML style
- `let l = 1 in ...` - allocates a new cell on the heap and sets the value to 1 (assume some trusted malloc exists).
- `assert(l == 1)` - if the assertion is true, then execution continues. If the assertion is false, then an assertion flag (a designated heap cell) is set to 1 and execution halts.

Example program

- High-level programs - ML style
- `let l = 1 in ...` - allocates a new cell on the heap and sets the value to 1 (assume some trusted malloc exists).
- `assert(l == 1)` - if the assertion is true, then execution continues. If the assertion is false, then an assertion flag (a designated heap cell) is set to 1 and execution halts.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example program

- High-level programs - ML style
- `let l = 1 in ...` - allocates a new cell on the heap and sets the value to 1 (assume some trusted malloc exists).
- `assert(l == 1)` - if the assertion is true, then execution continues. If the assertion is false, then an assertion flag (a designated heap cell) is set to 1 and execution halts.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Lemma

Given any program `adv`, `f(adv)` either runs forever, ends up in the *failed* configuration, or halts in a configuration where the assertion flag is 0.

Road map

Capability Machine

Formalisation

Example program

Logical Relation

Example revisited

Current work

Logical Relation - What is it

Logical relations in general

- Strong proof method

Logical Relation - What is it

Logical relations in general

- Strong proof method
- Used to show properties about programs

Logical Relation - What is it

Logical relations in general

- Strong proof method
- Used to show properties about programs
- Designed such that any program in the relation has a certain property

Logical Relation - What is it

Logical relations in general

- Strong proof method
- Used to show properties about programs
- Designed such that any program in the relation has a certain property
- Can be used when a direct proof does not suffice

Logical Relation - What is it

Logical relations in general

- Strong proof method
- Used to show properties about programs
- Designed such that any program in the relation has a certain property
- Can be used when a direct proof does not suffice
  - e.g., strong normalisation for STLC

Logical Relation - What is it

Logical relations in general

- Strong proof method
- Used to show properties about programs
- Designed such that any program in the relation has a certain property
- Can be used when a direct proof does not suffice
  - e.g., strong normalisation for STLC
- Can be used to reason about programs written in "real" programming languages

Logical Relation - What is it

Logical relations in general

- Strong proof method
- Used to show properties about programs
- Designed such that any program in the relation has a certain property
- Can be used when a direct proof does not suffice
  - e.g., strong normalisation for STLC
- Can be used to reason about programs written in "real" programming languages
- Extensional - not interested in what happens doing the execution, only interested in the result

Logical Relation

What we hope to achieve

- Any program will respect the limitations of the capability system.

Logical Relation

The property of this logical relation

Logical Relation

The property of this logical relation

- Any capability such that

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration, or
    - halt where the heap still satisfies the invariants

Logical Relation

The property of this logical relation

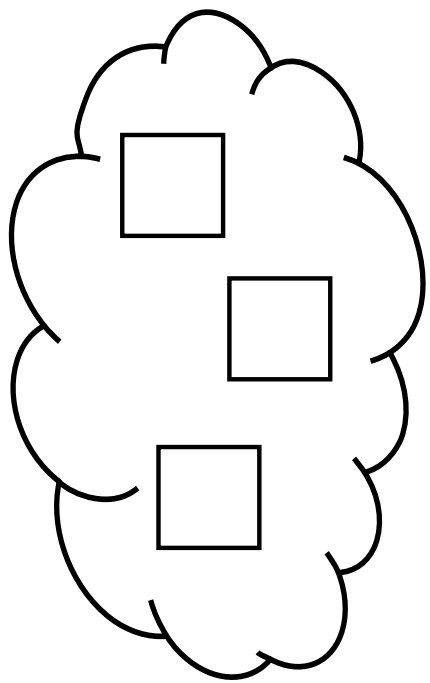
- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration, or
    - halt where the heap still satisfies the invariants

Logical Relation - Worlds, modelling the heap

World

- Collection of regions with invariants (e.g. $h(27) \mapsto 5$)

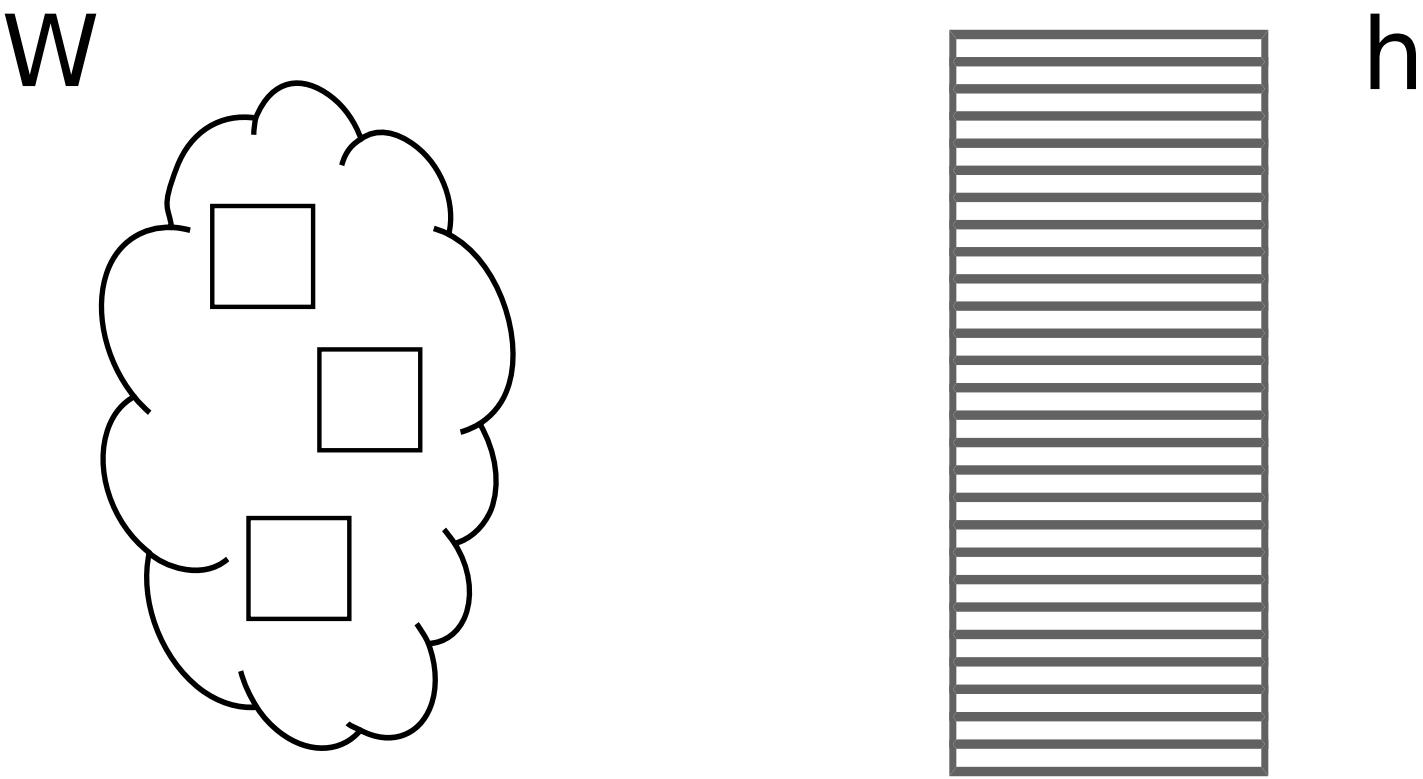
W



Logical Relation - Worlds, modelling the heap

World

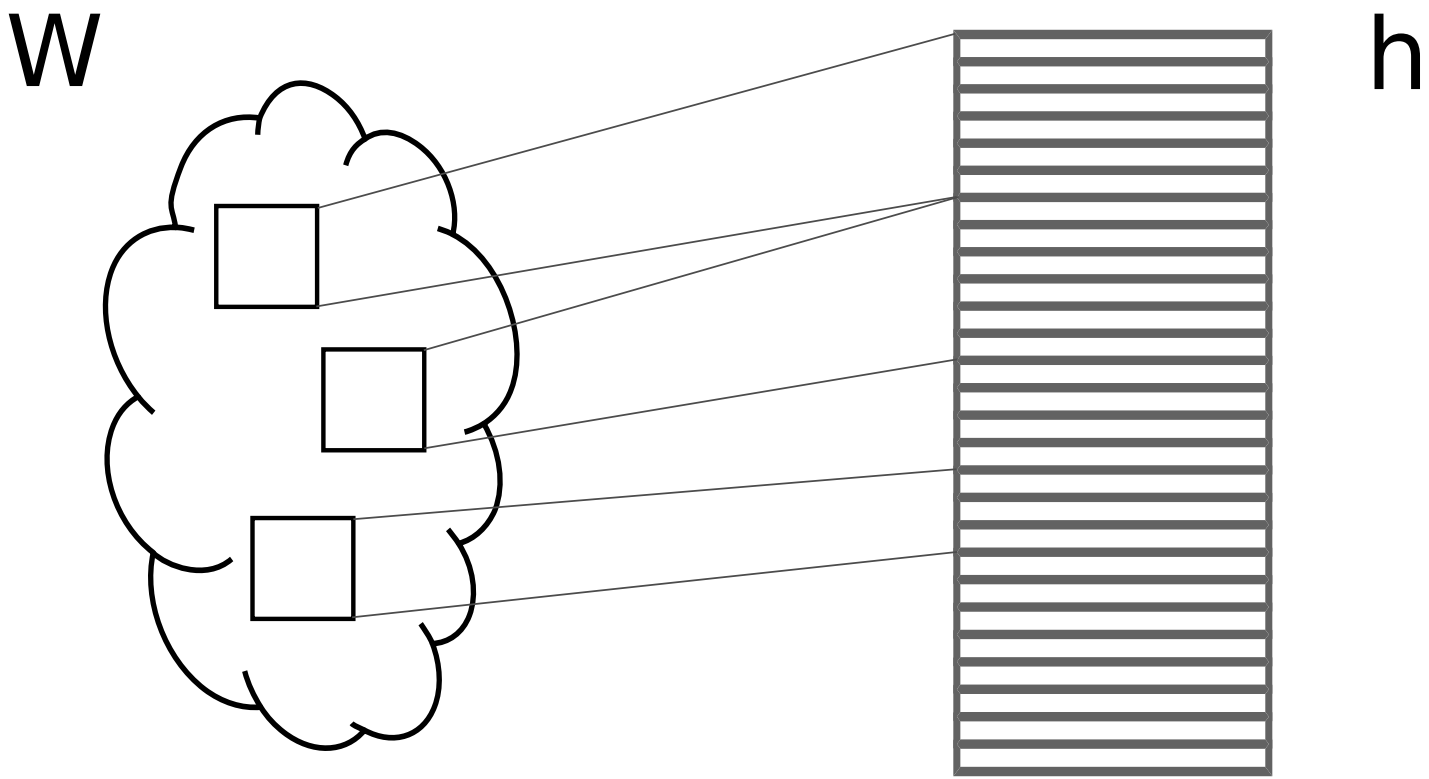
- Collection of regions with invariants (e.g. $h(27) \mapsto 5$)
- Model of the heap



Logical Relation - Worlds, modelling the heap

Heap satisfaction

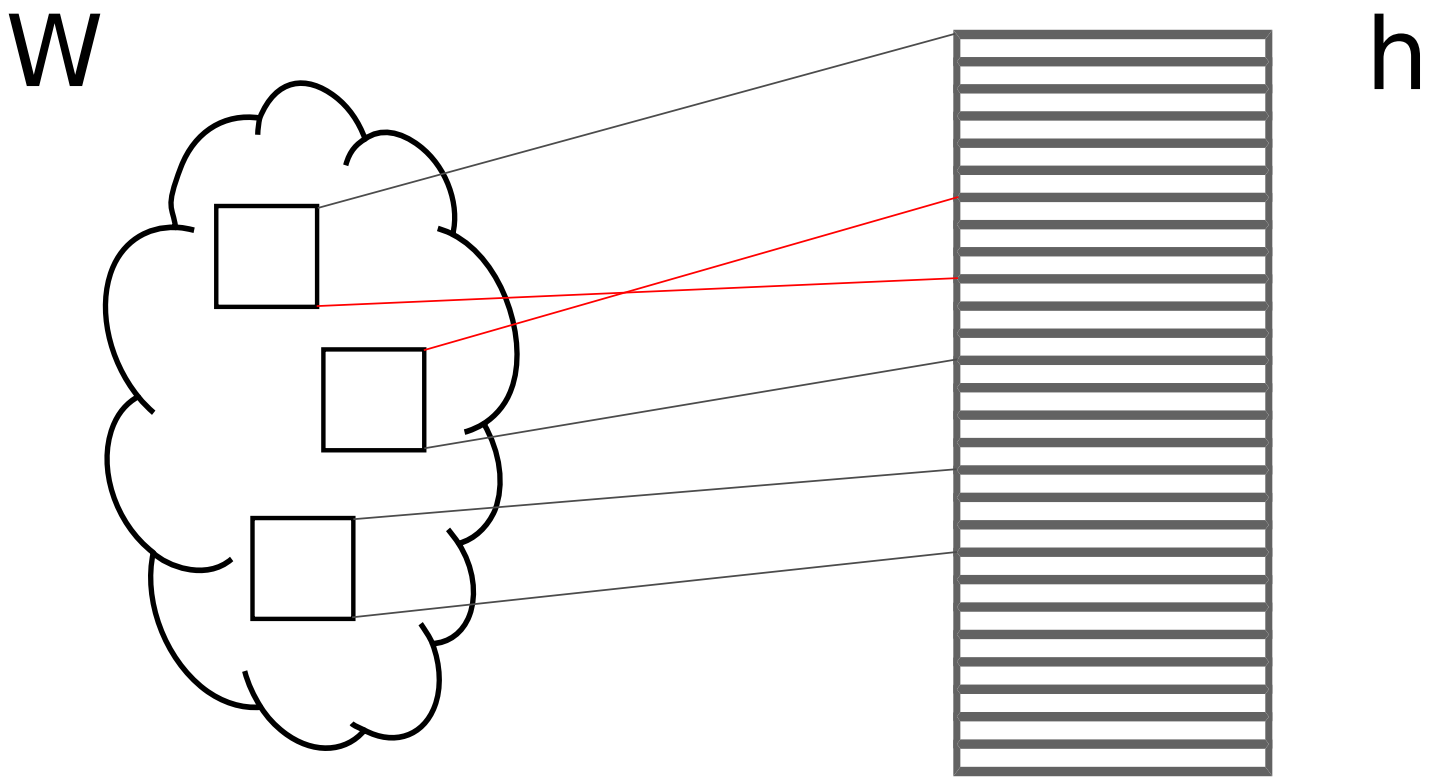
- Regions model parts of the heap



Logical Relation - Worlds, modelling the heap

Heap satisfaction

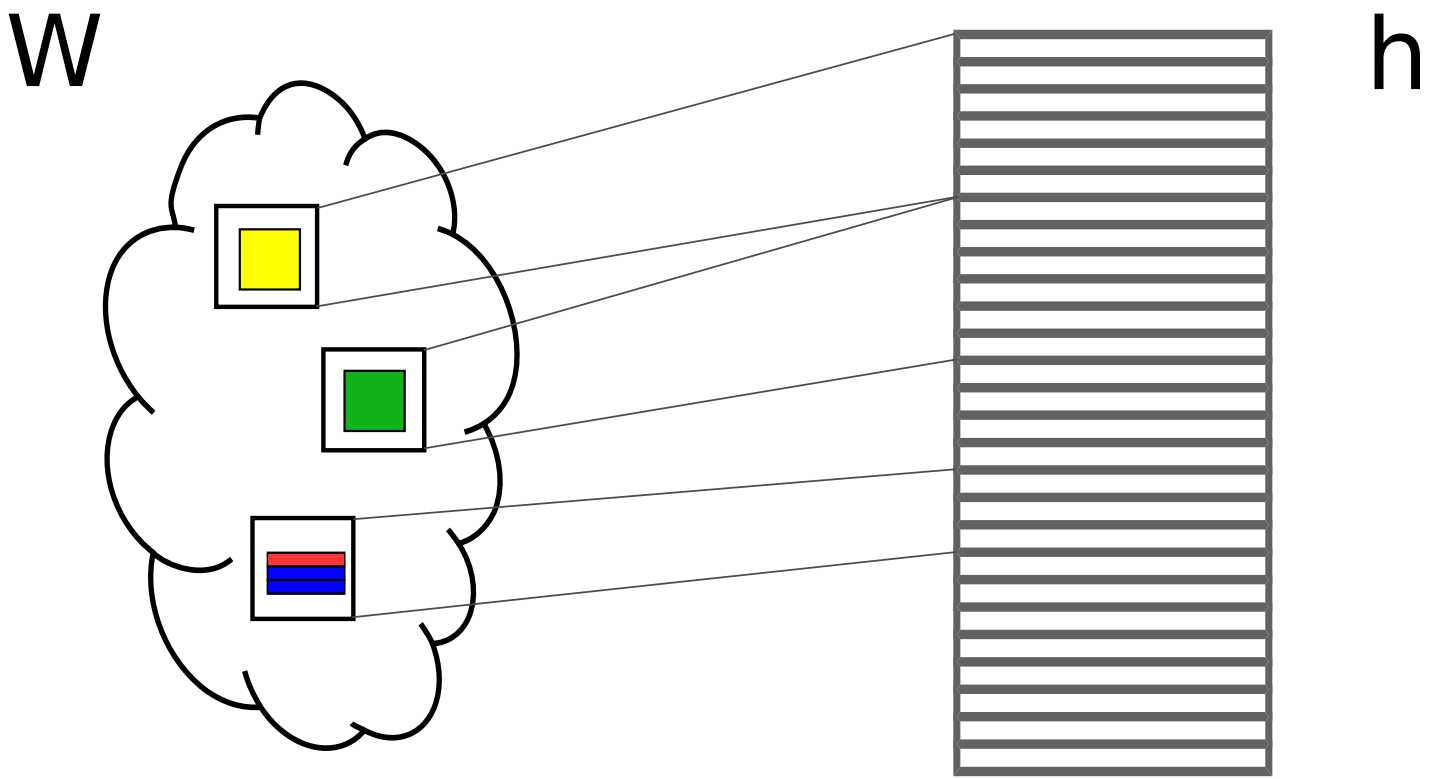
- Regions model parts of the heap
- Non-overlapping



Logical Relation - Worlds, modelling the heap

Heap satisfaction

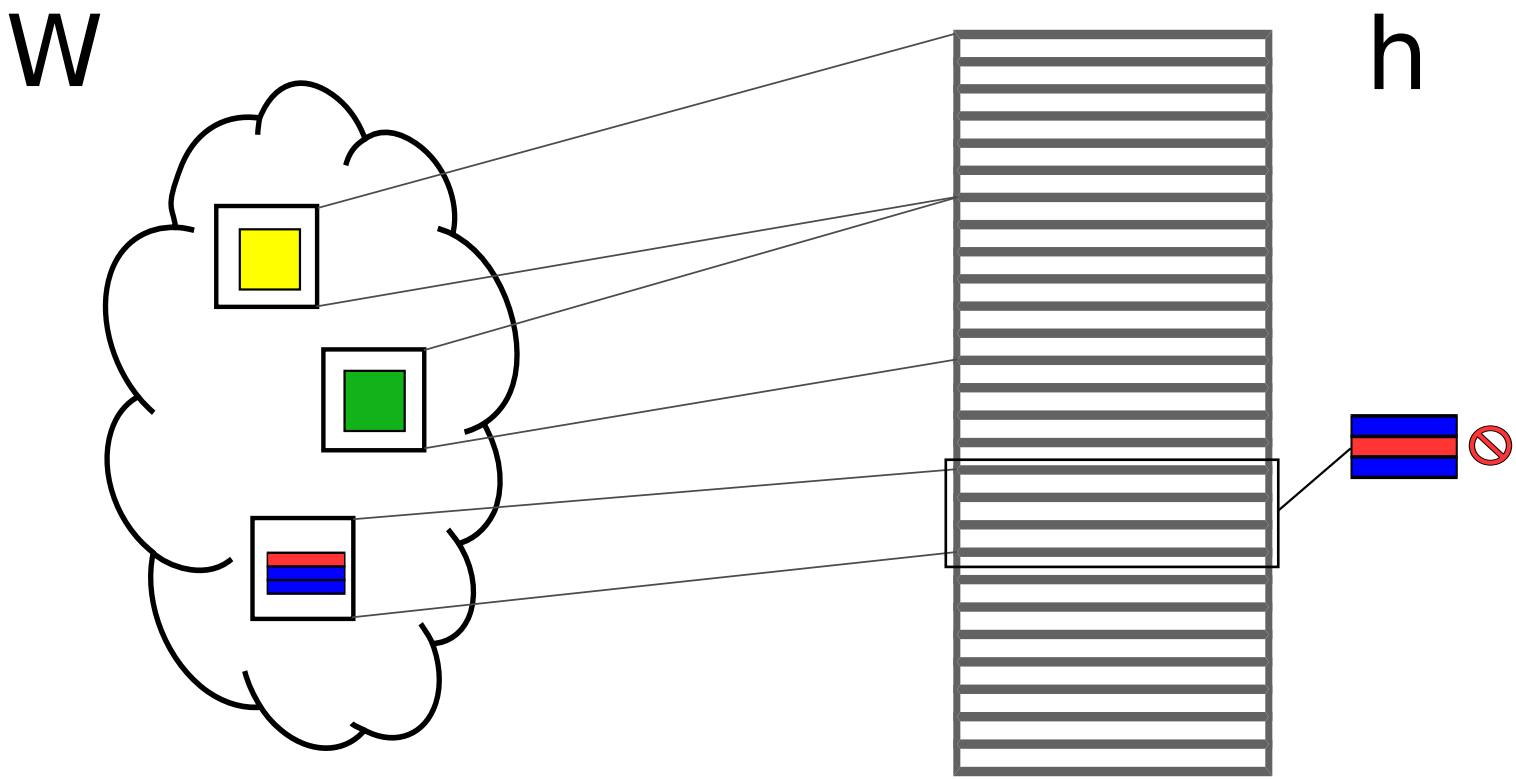
- Regions model parts of the heap
- Non-overlapping



Logical Relation - Worlds, modelling the heap

Heap satisfaction

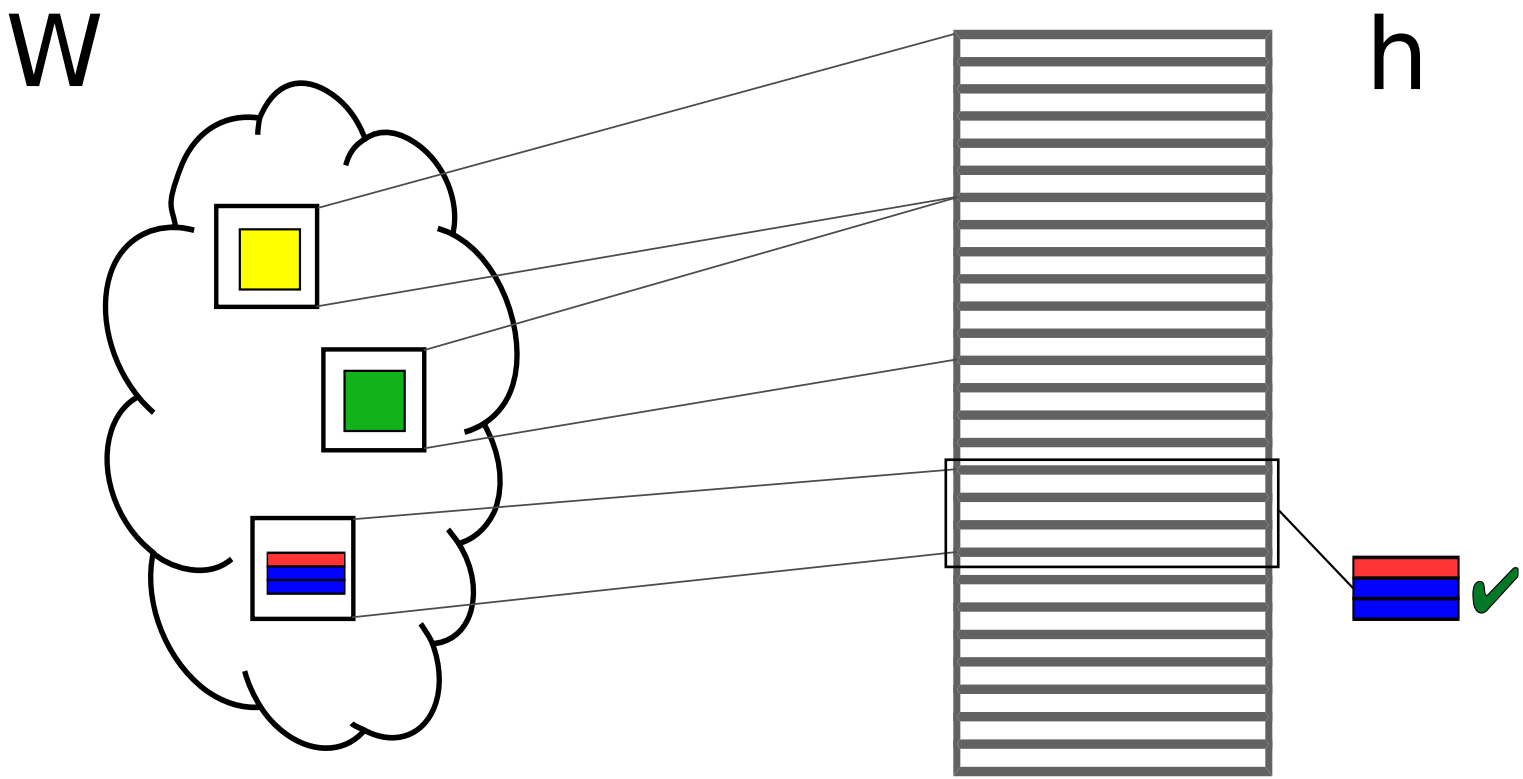
- Regions model parts of the heap
- Non-overlapping



Logical Relation - Worlds, modelling the heap

Heap satisfaction

- Regions model parts of the heap
- Non-overlapping
- $h : W$

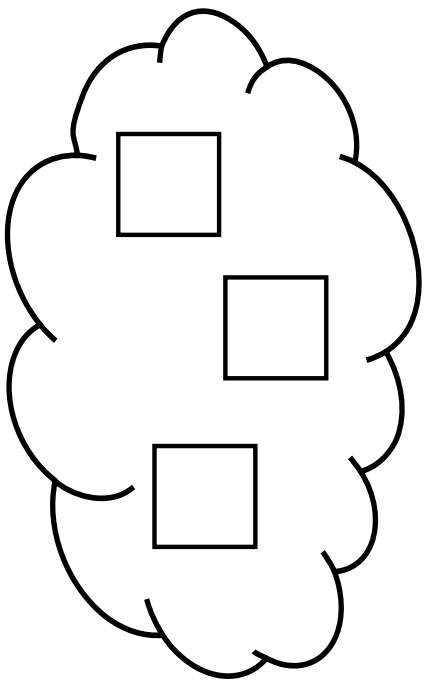


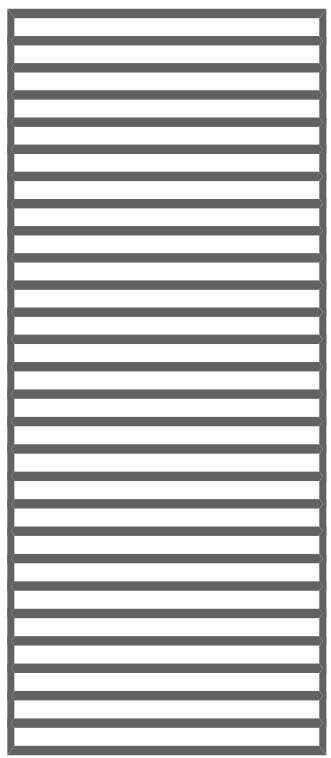
Logical Relation - Worlds, modelling the heap

Future World

- Heap changes over time, worlds have to cope with this:
- W
  - Same regions as before

W



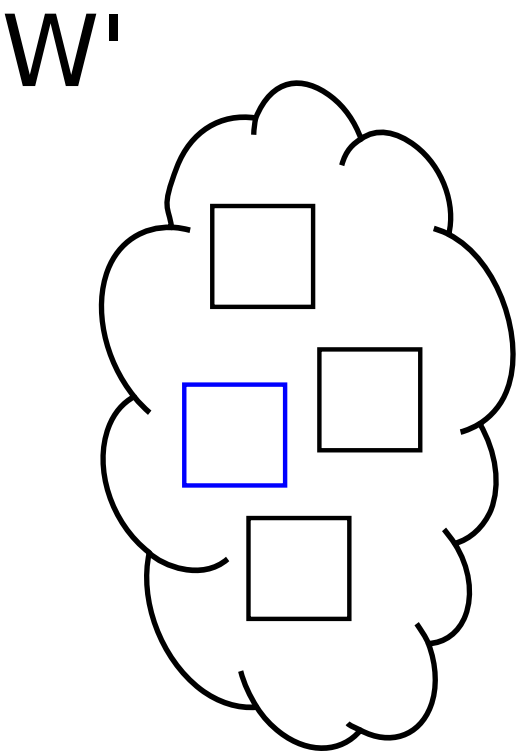


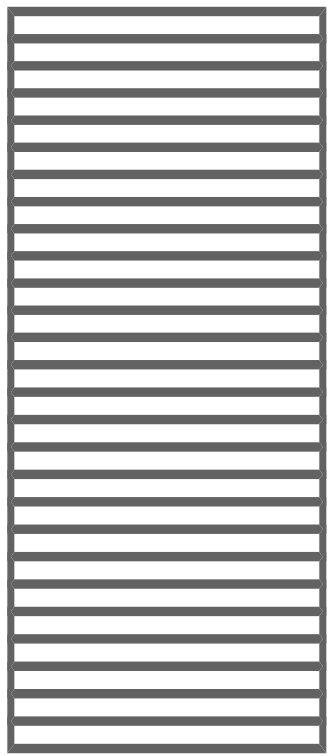
h

Logical Relation - Worlds, modelling the heap

Future World

- Heap changes over time, worlds have to cope with this:
- $W' \sqsupseteq W$
  - Same regions as before
  - New region(s)

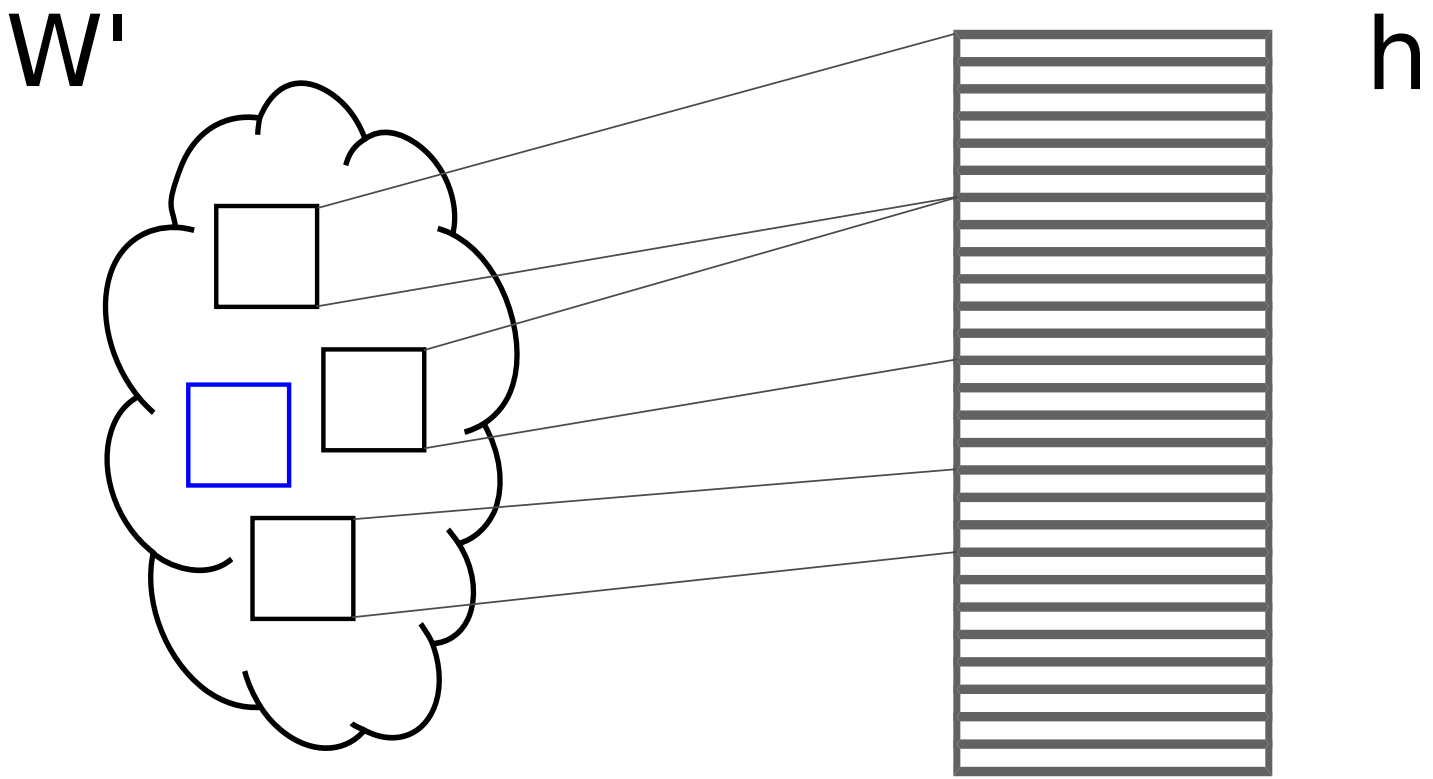




Logical Relation - Worlds, modelling the heap

Future World

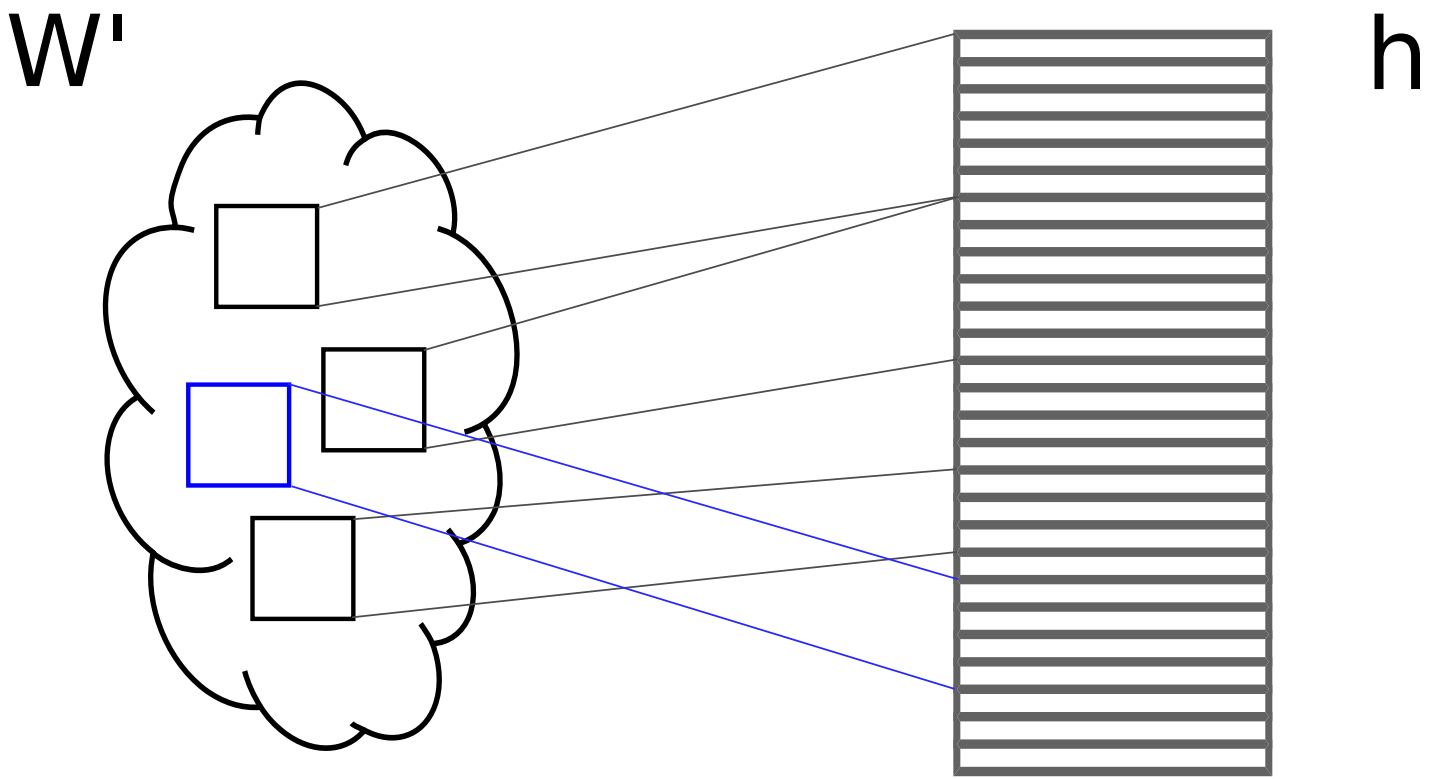
- Old regions model the same parts of the heap as before



Logical Relation - Worlds, modelling the heap

Future World

- Old regions model the same parts of the heap as before
- New part model new part of the heap



Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration, or
    - halt where the heap still satisfies the invariants

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration, or
    - halt where the heap still satisfies the invariants

Logical Relation - Observation relation

Property

- the execution will either
  - diverge,
  - end up in the *failed* configuration, or
  - halt where the heap still satisfies the invariants

$$\mathcal{O} \stackrel{def}{=} \lambda W.\, \{(reg, h) \mid$$

Logical Relation - Observation relation

Property

- the execution will either
  - diverge,
  - end up in the *failed* configuration, or
  - **halt where the heap still satisfies the invariants**

$$\mathcal{O} \stackrel{def}{=} \lambda W.\,\{(reg, h) \mid (\forall h'.\,(reg, h) \rightarrow^* (halted, h') \Rightarrow \exists W' \sqsupseteq W.\,h' : W') \vee$$

Logical Relation - Observation relation

Property

- the execution will either
  - **diverge**,
  - end up in the *failed* configuration, or
  - halt where the heap still satisfies the invariants

$$\mathcal{O} \stackrel{def}{=} \lambda W.\,\{(reg, h) \mid\ (\forall h'.\,(reg, h) \rightarrow^{*} (halted, h') \Rightarrow \exists W' \sqsupseteq W.\,h' : W') \vee (reg, h) \Downarrow \vee$$

Logical Relation - Observation relation

Property

- the execution will either
  - diverge,
  - **end up in the failed configuration**, or
  - halt where the heap still satisfies the invariants

$$\mathcal{O} \stackrel{def}{=} \lambda W.\,\{(reg, h) \mid (\forall h'.\,(reg, h) \rightarrow^* (halted, h') \Rightarrow \exists W' \sqsupseteq W.\,h' : W') \vee (reg, h) \Downarrow \vee (reg, h) \rightarrow^* failed\}$$

Logical Relation - Observation relation

Property

- the execution will either
  - diverge,
  - end up in the *failed* configuration, or
  - halt where the heap still satisfies the invariants

$$\mathcal{O} \stackrel{def}{=} \lambda W.\,\{(reg, h) \mid\ (\forall h'.\,(reg, h) \rightarrow^* (halted, h') \Rightarrow \exists W' \sqsupseteq W.\,h' : W') \vee (reg, h) \Downarrow \vee (reg, h) \rightarrow^* failed\}$$

Logical Relation - Observation relation

Property

- the execution will either
  - diverge,
  - end up in the *failed* configuration, or
  - halt where the heap still satisfies the invariants

$$\mathcal{O} \stackrel{def}{=} \lambda W.\,\{(reg, h) \mid (\forall h'.\,(reg, h) \rightarrow^* (halted, h') \Rightarrow \exists W' \sqsupseteq W.\,h' : W')\}$$

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration, or
    - halt where the heap still satisfies the invariants

Logical Relation

The property of this logical relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either
    - diverge,
    - end up in the *failed* configuration, or
    - halt where the heap still satisfies the invariants

Logical Relation - Expression relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - the execution will either ...

$$\mathcal{E} \stackrel{def}{=} \lambda W. \{c \mid$$

Logical Relation - Expression relation

- Any capability such that
  - **when executed in a "well-behaved" register-file**, and
  - a heap that satisfies certain invariants, then
  - the execution will either ...

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

Logical Relation - Expression relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - **a heap that satisfies certain invariants**, then
  - the execution will either ...

$$\mathcal{E} \stackrel{def}{=} \lambda W.\, \{c \mid \forall reg \in \mathcal{R}(W).$$
$$\forall h : W.$$

Logical Relation - Expression relation

- Any capability such that
  - when executed in a "well-behaved" register-file, and
  - a heap that satisfies certain invariants, then
  - **the execution will either ...**

$$\mathcal{E} \stackrel{def}{=} \lambda W.\, \{c \mid \forall reg \in \mathcal{R}(W).\ \forall h : W.\ (reg[\mathrm{pc} \mapsto c], h) \in \mathcal{O}(W)\}$$

Logical Relation - Register-file relation

"Well-behaved" register-file

$$\mathcal{R} \stackrel{def}{=} \lambda W.\,\{reg \mid$$

Logical Relation - Register-file relation

"Well-behaved" register-file

- All registers but the pc-register

$$\mathcal{R} \stackrel{def}{=} \lambda W.\, \{reg \mid \forall r \in \mathrm{RegisterName} \setminus \{\mathrm{pc}\}.$$

Logical Relation - Register-file relation

"Well-behaved" register-file

- All registers but the pc-register
  - pc was overwritten in the $\mathcal{E}$ anyway

$$\mathcal{R} \stackrel{def}{=} \lambda W.\, \{reg \mid \forall r \in \mathrm{RegisterName} \setminus \{\mathrm{pc}\}.$$

Logical Relation - Register-file relation

"Well-behaved" register-file

- All registers but the pc-register
  - pc was overwritten in the $\mathcal{E}$ anyway
- should contain a "well-behaved" word

$$\mathcal{R} \stackrel{def}{=} \lambda W.\, \{reg \mid \forall r \in \mathrm{RegisterName} \setminus \{\mathrm{pc}\}.\ reg(r) \in \mathcal{V}(W)\}$$

Logical Relation - Value relation

"Well-behaved words"

$$
\begin{aligned}
\mathcal{V} \stackrel{def}{=} \lambda\, W.\ & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{(\mathrm{o}, \mathit{base}, \mathit{end}, a)\} \cup \\
& \{(\mathrm{ro}, \mathit{base}, \mathit{end}, a) \mid \qquad\qquad \} \cup \\
& \{(\mathrm{rw}, \mathit{base}, \mathit{end}, a) \mid \qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \cup \\
& \{(\mathrm{rx}, \mathit{base}, \mathit{end}, a) \mid \qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \} \cup \\
& \{(\mathrm{e}, \mathit{base}, \mathit{end}, a) \mid \qquad\qquad\quad \cup \\
& \{(\mathrm{rwx}, \mathit{base}, \mathit{end}, a) \mid \qquad\qquad\quad \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \}
\end{aligned}
$$

Logical Relation - Value relation

"Well-behaved words"

$$
\begin{aligned}
\mathcal{V} \stackrel{def}{=} \lambda\, W.\ & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{(\mathrm{o}, \mathit{base}, \mathit{end}, a)\} \cup \\
& \{(\mathrm{ro}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rw}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \cup \\
& \{(\mathrm{rx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \cup \\
& \{(\mathrm{e}, \mathit{base}, \mathit{end}, a) \mid \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup \\
& \{(\mathrm{rwx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \}
\end{aligned}
$$

Logical Relation - Value relation

"Well-behaved words"

$$
\begin{aligned}
\mathcal{V} \stackrel{def}{=} \lambda\, W.\ & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{(\mathrm{o}, \mathit{base}, \mathit{end}, a)\} \cup \\
& \{(\mathrm{ro}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rw}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{writeCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \}\cup \\
& \{(\mathrm{e}, \mathit{base}, \mathit{end}, a) \mid \qquad \cup \\
& \{(\mathrm{rwx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{writeCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \}
\end{aligned}
$$

Logical Relation - Value relation

"Well-behaved words"

$$
\begin{aligned}
\mathcal{V} \stackrel{def}{=} \lambda\, W.\ & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{(\mathrm{o}, \mathit{base}, \mathit{end}, a)\} \cup \\
& \{(\mathrm{ro}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rw}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{writeCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{executeCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{e}, \mathit{base}, \mathit{end}, a) \mid \qquad \cup \\
& \{(\mathrm{rwx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{writeCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{executeCondition}(\mathit{base}, \mathit{end}, W)\}
\end{aligned}
$$

Logical Relation - Value relation

"Well-behaved words"

$$
\begin{aligned}
\mathcal{V} \stackrel{def}{=} \lambda\, W.\ & \{i \mid i \in \mathbb{Z}\} \cup \\
& \{(\mathrm{o}, \mathit{base}, \mathit{end}, a)\} \cup \\
& \{(\mathrm{ro}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rw}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{writeCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{rx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{executeCondition}(\mathit{base}, \mathit{end}, W)\} \cup \\
& \{(\mathrm{e}, \mathit{base}, \mathit{end}, a) \mid \mathit{enterCondition}(\mathit{base}, \mathit{end}, a, W) \cup \\
& \{(\mathrm{rwx}, \mathit{base}, \mathit{end}, a) \mid \mathit{readCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{writeCondition}(\mathit{base}, \mathit{end}, W) \wedge \\
& \qquad \mathit{executeCondition}(\mathit{base}, \mathit{end}, W)\}
\end{aligned}
$$

Logical Relation - Execute and enter conditions

Execution condition

$$executeCondition(base, end, W) \stackrel{def}{=}$$

Logical Relation - Execute and enter conditions

Execution condition

- May be used at any point in the future

$$executeCondition(base, end, W) \stackrel{def}{=}$$
$$\forall W' \sqsupseteq W.$$

Logical Relation - Execute and enter conditions

Execution condition

- May be used at any point in the future
- Can be executed from any address in the range of authority

$$
\begin{aligned}
&\mathit{executeCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=} \\
&\quad \forall W' \sqsupseteq W. \\
&\qquad \forall a \in [\mathit{base}, \mathit{end}].
\end{aligned}
$$

Logical Relation - Execute and enter conditions

Execution condition

- May be used at any point in the future
- Can be executed from any address in the range of authority
- Should produce a "well-behaved" result, i.e., it should be in the $\mathcal{E}$-relation

$$
\begin{aligned}
&\mathit{executeCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=} \\
&\quad \forall W' \sqsupseteq W. \\
&\quad\quad \forall a \in [\mathit{base}, \mathit{end}]. \\
&\quad\quad\quad (\mathrm{rx}, \mathit{base}, \mathit{end}, a) \in \mathcal{E}(W')
\end{aligned}
$$

Logical Relation - Execute and enter conditions

Enter condition

$$enterCondition(base, end, a, W) \stackrel{def}{=}$$

Logical Relation - Execute and enter conditions

Enter condition

- May be used at any point in the future

$$enterCondition(base, end, a, W) \stackrel{def}{=}$$
$$\forall W' \sqsupseteq W.$$

Logical Relation - Execute and enter conditions

Enter condition

- May be used at any point in the future
- Can only be executed from the specified address

$$enterCondition(base, end, a, W) \stackrel{def}{=}$$
$$\forall W' \sqsupseteq W.$$

Logical Relation - Execute and enter conditions

Enter condition

- May be used at any point in the future
- Can only be executed from the specified address
- Should produce a "well-behaved" result, i.e., it should be in the $\mathcal{E}$-relation

$$
\begin{aligned}
&\mathit{enterCondition}(\mathit{base}, \mathit{end}, a, W) \stackrel{\mathit{def}}{=} \\
&\quad \forall W' \sqsupseteq W. \\
&\qquad (\mathrm{rx}, \mathit{base}, \mathit{end}, a) \in \mathcal{E}(W')
\end{aligned}
$$

Logical Relation - Read and write conditions

Read condition

- World models heap, so it describes what we might read

$$readCondition(base, end, W) \stackrel{def}{=}$$

Logical Relation - Read and write conditions

Read condition

- World models heap, so it describes what we might read
- Some region should govern the part of the heap we can read from

$$\mathit{readCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=}$$
$$\exists r \in \mathrm{RegionName}.$$

Logical Relation - Read and write conditions

Read condition

- World models heap, so it describes what we might read
- Some region should govern the part of the heap we can read from
- The region may govern a larger part of the heap

$$\begin{aligned} &\mathit{readCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=} \\ &\quad \exists r \in \mathrm{RegionName}. \\ &\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \end{aligned}$$

Logical Relation - Read and write conditions

Read condition

- The region should be subset of the standard region $\iota_{base',end'}$

$$
\begin{aligned}
&\mathit{readCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=} \\
&\quad \exists r \in \mathrm{RegionName}. \\
&\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
&\qquad\quad W(r) \subseteq \iota_{\mathit{base}',\mathit{end}'}
\end{aligned}
$$

Logical Relation - Read and write conditions

Read condition

- The region should be subset of the standard region $\iota_{base',end'}$

$$
\begin{aligned}
&readCondition(base, end, W) \stackrel{def}{=} \\
&\quad \exists r \in \mathrm{RegionName}. \\
&\qquad \exists [base', end'] \supseteq [base, end]. \\
&\qquad\quad W(r) \subseteq \iota_{base',end'}
\end{aligned}
$$

- $\iota_{base',end'}$ is a standard region that requires

Logical Relation - Read and write conditions

Read condition

- The region should be subset of the standard region $\iota_{base',end'}$

$$
\begin{aligned}
&readCondition(base, end, W) \stackrel{def}{=} \\
&\quad \exists r \in \mathrm{RegionName}. \\
&\qquad \exists [base', end'] \supseteq [base, end]. \\
&\qquad\quad W(r) \subseteq \iota_{base',end'}
\end{aligned}
$$

- $\iota_{base',end'}$ is a standard region that requires
  - Range of heap segment to be $[base', end']$

Logical Relation - Read and write conditions

Read condition

- The region should be subset of the standard region $\iota_{base',end'}$

$$
\begin{aligned}
&readCondition(base, end, W) \stackrel{def}{=} \\
&\quad \exists r \in \mathrm{RegionName}. \\
&\qquad \exists [base', end'] \supseteq [base, end]. \\
&\qquad\quad W(r) \subseteq \iota_{base',end'}
\end{aligned}
$$

- $\iota_{base',end'}$ is a standard region that requires
  - Range of heap segment to be $[base', end']$
  - All the words in the heap segment should be in the $\mathcal{V}$-relation

Logical Relation - Read and write conditions

Read condition

- The region should be subset of the standard region $\iota_{base',end'}$
- Intuition:
  - If untrusted code got this capability, then it should only be able to read "well-behaved" words.

$$\begin{aligned} &\mathit{readCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=} \\ &\quad \exists r \in \mathrm{RegionName}. \\ &\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\ &\qquad\quad W(r) \subseteq \iota_{\mathit{base}',\mathit{end}'} \end{aligned}$$

- $\iota_{base',end'}$ is a standard region that requires
  - Range of heap segment to be $[base', end']$
  - All the words in the heap segment should be in the $\mathcal{V}$-relation

Logical Relation - Read and write conditions

Write condition

- World should describe what we are allowed write

$$writeCondition(base, end, W) \stackrel{def}{=}$$

- $\iota_{base',end'}$ is a standard region that requires
  - Range of heap segment to be $[base', end']$
  - All the words in the heap segment should be in the $\mathcal{V}$-relation

Logical Relation - Read and write conditions

Write condition

- World should describe what we are allowed write
- Some region governs the part of the heap we may write to

$$\begin{aligned} &\mathit{writeCondition}(\mathit{base}, \mathit{end}, W) \stackrel{\mathit{def}}{=} \\ &\quad \exists r \in \mathrm{RegionName}. \\ &\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \end{aligned}$$

- $\iota_{\mathit{base}',\mathit{end}'}$ is a standard region that requires
  - Range of heap segment to be $[\mathit{base}', \mathit{end}']$
  - All the words in the heap segment should be in the $\mathcal{V}$-relation

Logical Relation - Read and write conditions

Write condition

- The region should be *superset* of the standard region $\iota_{base',end'}$

$$
\begin{aligned}
&writeCondition(base, end, W) \stackrel{def}{=} \\
&\quad \exists r \in \mathrm{RegionName}. \\
&\qquad \exists [base', end'] \supseteq [base, end]. \\
&\qquad\quad W(r) \supseteq \iota_{base',end'}
\end{aligned}
$$

- $\iota_{base',end'}$ is a standard region that requires
  - Range of heap segment to be $[base', end']$
  - All the words in the heap segment should be in the $\mathcal{V}$-relation

Logical Relation - Read and write conditions

Write condition

- The region should be *superset* of the standard region $\iota_{base',end'}$
- Intuition:
  - If untrusted code got this capability, then it can at least write something well-behaved, but also other things.

$$\begin{aligned} &writeCondition(base, end, W) \stackrel{def}{=} \\ &\exists r \in \mathrm{RegionName}. \\ &\quad \exists [base', end'] \supseteq [base, end]. \\ &\quad\quad W(r) \supseteq \iota_{base',end'} \end{aligned}$$

- $\iota_{base',end'}$ is a standard region that requires
  - Range of heap segment to be $[base', end']$
  - All the words in the heap segment should be in the $\mathcal{V}$-relation

Fundamental theorem of logical relations

Lemma (FTLR)

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

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

Fundamental theorem of logical relations

Lemma (FTLR)

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

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

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

Fundamental theorem of logical relations

Lemma (FTLR)

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

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

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

Fundamental theorem of logical relations

Lemma (FTLR)

For all $W \in \mathrm{World}$, *perm* $\in \mathrm{Perm}$, *and* $\mathit{base}, \mathit{end}, a \in \mathrm{Addr}$,

if

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

or

$$\mathit{perm} = \mathrm{rwx} \text{ and } \mathit{read\text{-}/writeCond}(W, \mathit{base}, \mathit{end})$$

then

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

Road map

Example: local state revisited

Lemma

Given any program `adv`, `f(adv)` either runs forever, ends up in the *failed* configuration, or halts in a configuration where the assertion flag is 0.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`
- Define world with the following regions:

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`
- Define world with the following regions:
  - `f` code remains unchanged

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`
- Define world with the following regions:
  - `f` code remains unchanged
  - `l` remains 1

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`
- Define world with the following regions:
  - `f` code remains unchanged
  - `l` remains 1
  - standard region governs `adv`

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`
- Define world with the following regions:
  - `f` code remains unchanged
  - `l` remains 1
  - standard region governs `adv`
  - assertion flag is 0

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch

- Assuming `adv` is only code and given as enter capability
- Run program until just after the jump to `adv`
- Define world with the following regions:
  - `f` code remains unchanged
  - `l` remains 1
  - standard region governs `adv`
  - assertion flag is 0
- Use FTLR on `adv` capability

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Use FTLR on adv capability

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Use FTLR on `adv` capability
- By design, the heap satisfies the world

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Use FTLR on adv capability
- By design, the heap satisfies the world
- Register-file in $\mathcal{R}$-relation:

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Use FTLR on adv capability
- By design, the heap satisfies the world
- Register-file in $\mathcal{R}$-relation:
  - All registers but two contain 0, so trivial.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Use FTLR on adv capability
- By design, the heap satisfies the world
- Register-file in $\mathcal{R}$-relation:
  - All registers but two contain 0, so trivial.
  - One is pc-register, so we don't care about it.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Use FTLR on adv capability
- By design, the heap satisfies the world
- Register-file in $\mathcal{R}$-relation:
  - All registers but two contain 0, so trivial.
  - One is pc-register, so we don't care about it.
  - The other is the continuation (passed as enter capability), so *enterCondition* must hold

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- World highlights:
  - `f` code remains unchanged
  - `l` remains 1
  - assertion flag is 0
- The continuation satisfies *enterCondition*:

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- World highlights:
  - `f` code remains unchanged
  - `l` remains 1
  - assertion flag is 0
- The continuation satisfies *enterCondition*:
  - In a future world, the continuation must be in $\mathcal{E}$

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- World highlights:
  - `f` code remains unchanged
  - `l` remains 1
  - assertion flag is 0
- The continuation satisfies *enterCondition*:
  - In a future world, the continuation must be in $\mathcal{E}$
  - Executing from continuation, `l` is still 1, so assertion does not fail.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- World highlights:
  - `f` code remains unchanged
  - `l` remains 1
  - assertion flag is 0
- The continuation satisfies *enterCondition*:
  - In a future world, the continuation must be in $\mathcal{E}$
  - Executing from continuation, `l` is still 1, so assertion does not fail.
  - Execution halts and assertion flag is 0

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Backtracking a lot, we have just shown that the register-file was in the $\mathcal{R}$-relation

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Proof sketch (continued)

- Backtracking a lot, we have just shown that the register-file was in the $\mathcal{R}$-relation
- By $\texttt{adv} \in \mathcal{E}$: execution diverges, fails, or terminates without the assertion failing.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Example: local state revisited

Lemma

Given any program `adv`, `f(adv)` either runs forever, ends up in the *failed* configuration, or halts in a configuration where the assertion flag is 0.

```
let f = fun adv =>
          let l = 1 in
          adv();
          assert (l == 1)
```

Road map

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?
  - No,

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?
  - No, `adv` may save the continuation from the first call

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?
  - No, `adv` may save the continuation from the first call

Local capabilities

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?
  - No, `adv` may save the continuation from the first call

Local capabilities

- *local*/*global* capabilities

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?
  - No, `adv` may save the continuation from the first call

Local capabilities

- *local*/*global* capabilities
- *permit write local* capabilities

What are we working on now?

```
let f = fun adv =>
  let l = 0 in
  adv();
  assert(l == 0);
  l := 1;
  adv()
```

- Assuming standard calling convention, can we show that the assertion never fails?
  - No, `adv` may save the continuation from the first call

Local capabilities

- *local*/*global* capabilities
- *permit write local* capabilities
- *Local* capabilities can only be stored through *permit write local* capabilities

Questions?

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 Architecuture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.