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]](image-url)
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]](image)
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

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

\[ \text{Perm} \overset{\text{def}}{=} \{ \text{o}, \text{ro}, \text{rw}, \text{rx}, \text{e}, \text{rwx} \} \]
Permissions

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

\[
\text{Perm} \overset{\text{def}}{=} \{ \text{o}, \text{ro}, \text{rw}, \text{rx}, \text{e}, \text{rwx} \}
\]
Permissions

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

\[
\text{Perm} \overset{\text{def}}{=} \{ \text{o}, \text{ro}, \text{rw}, \text{rx}, \text{e}, \text{rwx} \}
\]
Formalisation - Permissions

Permissions

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

\[ \text{Perm} \overset{\text{def}}{=} \{ \text{o, ro, rw} \} \]
Permissions

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

\[
\text{Perm} \overset{\text{def}}{=} \{ \text{o, ro, rw, rx, } \}
\]
Formalisation - Permissions

Permissions

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

\[
\text{Perm} \overset{\text{def}}{=} \{ o, \text{ro}, \text{rw}, \text{rx}, e, \}\]
Permissions

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

\[
\text{Perm} \overset{\text{def}}{=} \{ o, ro, rw, rx, e, rwx \}
\]
Formalisation - Capabilities

Capability

\[
\text{Cap} \overset{\text{def}}{=} \text{Perm} \times \text{Addr} \times \text{Addr} \times \text{Addr}
\]

Example: \((e, 30, 42, 30)\)
Formalisation - Capabilities

Capability

- Permission

\[ \text{Cap} \overset{\text{def}}{=} \text{Perm} \times \text{Addr} \times \text{Addr} \times \text{Addr} \]

Example: \((e, 30, 42, 30)\)
Formalisation - Capabilities

Capability

▶ Permission

Cap \overset{\text{def}}{=} \text{Perm}
Formalisation - Capabilities

**Capability**

- Permission
- Range of authority

\[
\text{Cap} \overset{\text{def}}{=} \text{Perm}
\]
Formalisation - Capabilities

Capability
- Permission
- Range of authority

\[ \text{Addr} \overset{\text{def}}{=} \mathbb{N} \]

\[ \text{Cap} \overset{\text{def}}{=} \text{Perm} \]
Formalisation - Capabilities

**Capability**
- Permission
- Range of authority

\[
\text{Addr} \overset{\text{def}}{=} \mathbb{N}
\]

\[
\text{Cap} \overset{\text{def}}{=} \text{Perm} \times \text{Addr} \times \text{Addr}
\]
Formalisation - Capabilities

**Capability**

- Permission
- Range of authority
- Pointer

\[
\text{Addr} \overset{\text{def}}{=} \mathbb{N}
\]

\[
\text{Cap} \overset{\text{def}}{=} \text{Perm} \times \text{Addr} \times \text{Addr}
\]
Formalisation - Capabilities

Capability

- Permission
- Range of authority
- Pointer

\[
\text{Addr} \overset{\text{def}}{=} \mathbb{N}
\]

\[
\text{Cap} \overset{\text{def}}{=} \text{Perm} \times \text{Addr} \times \text{Addr} \times \text{Addr}
\]
Formalisation - Capabilities

Capability

▶ Permission
▶ Range of authority
▶ Pointer

\[ \text{Addr} \overset{\text{def}}{=} \mathbb{N} \]

\[ \text{Cap} \overset{\text{def}}{=} \text{Perm} \times \text{Addr} \times \text{Addr} \times \text{Addr} \]

Example: \((e, 30, 42, 30)\)
Formalisation - Words and register file

Words

Word $\overset{\text{def}}{=} \text{Word}$
Formalisation - Words and register file

**Words**
- Capability

\[
\text{Word} \overset{\text{def}}{=} \text{Capability}
\]
Formalisation - Words and register file

Words

- Capability

\[
\text{Word} \overset{\text{def}}{=} \text{Cap}
\]
Formalisation - Words and register file

**Words**

- Capability
- Data (and instructions)

\[
\text{Word} \overset{\text{def}}{=} \text{Cap}
\]
Formalisation - Words and register file

Words
- Capability
- Data (and instructions)

\[ \text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z} \]
Formalisation - Words and register file

Words

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

\[
\text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z}
\]
Formalisation - Words and register file

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

\[
\text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z}
\]

Register file

\[
\text{Reg} \overset{\text{def}}{=}
\]
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
  \[
  \text{Word} \overset{\text{def}}{=} \text{Cap} + \mathbb{Z}
  \]

**Register file**
- Assume finite set of registers RegisterName \( \ni \) pc
  \[
  \text{Reg} \overset{\text{def}}{=} \text{RegisterName} \rightarrow \text{Word}
  \]
Formalisation - Heap and configurations

**Heap**

$$\text{Heap} \overset{\text{def}}{=} \text{Map from } \text{Addr} \text{ to } \text{Word}$$
Formalisation - Heap and configurations

**Heap**

- Map from Addr to Word

\[
\text{Heap} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]
Formalisation - Heap and configurations

Heap
- Map from \( \text{Addr} \) to \( \text{Word} \)

\[
\text{Heap} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration

\[
\text{Conf} \overset{\text{def}}{=}
\]
Formalisation - Heap and configurations

Heap
- Map from Addr to Word

\[
\text{Heap} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration
- Executable configuration

\[
\text{Conf} \overset{\text{def}}{=} \quad
\]
Formalisation - Heap and configurations

Heap
- Map from Addr to Word

\[
\text{Heap} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration
- Executable configuration

\[
\text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Heap}
\]
Formalisation - Heap and configurations

**Heap**

- Map from Addr to Word

\[ \text{Heap} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word} \]

**Configuration**

- Executable configuration
- Successfully halted configuration

\[ \text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Heap} \]
Formalisation - Heap and configurations

**Heap**
- Map from Addr to Word
  \[
  \text{Heap} \overset{\text{def}}{=} \text{Addr} \to \text{Word}
  \]

**Configuration**
- Executable configuration
- Successfully halted configuration

\[
\text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Heap} + \{\text{halted}\} \times \text{Heap}
\]
Formalisation - Heap and configurations

Heap
- Map from Addr to Word

\[
\text{Heap} \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\]

Configuration
- Executable configuration
- Successfully halted configuration
- Failed configuration

\[
\text{Conf} \overset{\text{def}}{=} \text{Reg} \times \text{Heap} + \{\text{failed}\} + \{\text{halted}\} \times \text{Heap}
\]
Formalisation - Instructions

Syntax

Instructions ::=
Formalisation - Instructions

Syntax

\[ rn ::= n \mid r \]

Instructions ::=
Formalisation - Instructions

Syntax

- The normal instructions

\[ \text{rn} ::= n \mid r \]

Instructions ::=
Formalisation - Instructions

Syntax

▶ The normal instructions

\[ r_n ::= n \mid r \]

Instructions ::= jmp \( r \) \mid jnz \( r \ r_n \) \mid move \( r \ r_n \) \mid load \( r \ r \) \mid store \( r \ r \) \mid plus \( r \ r_n \ r_n \)
Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

\[
\begin{align*}
    rn & ::= n \mid r \\
    \text{Instructions} & ::= \text{jmp } r \mid \text{jnz } r \text{ rn} \mid \text{move } r \text{ rn} \mid \\
                     & \quad \text{load } r \text{ r} \mid \text{store } r \text{ r} \mid \text{plus } r \text{ rn} \text{ rn}
\end{align*}
\]
Formalisation - Instructions

Syntax

- The normal instructions
- The capability manipulation instructions

\[
\begin{align*}
\text{rn} & : = \ n \mid r \\
\text{Instructions} & : = \ \text{jmp} \ r \mid \text{jnz} \ r \ r n \mid \text{move} \ r \ r n \mid \\
& \quad \text{load} \ r \ r \mid \text{store} \ r \ r \mid \text{plus} \ r \ r n \ r n \mid \\
& \quad \text{lea} \ r \ r n \mid \text{restrict} \ r \ r \ r n \mid \\
& \quad \text{subseg} \ r \ r n \ r n
\end{align*}
\]
Formalisation - Instructions

Syntax

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

\[ rn ::= n \mid r \]

Instructions ::= \( \text{jmp } r \mid \text{jnz } r \text{ } \)rn\mid \text{move } r \text{ } \)rn\mid \text{load } r \text{ } \)r\mid \text{store } r \text{ } \)r\mid \text{plus } r \text{ } \)rn\text{ } \)rn\mid \text{lea } r \text{ } \)rn\mid \text{restrict } r \text{ } \)r\text{ } \)rn\mid \text{subseg } r \text{ } \)rn\text{ } \)rn
Formalisation - Instructions

Syntax

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

\[ rn ::= n \mid r \]

Instructions ::= jmp r | jnz r rn | move r rn |
load r r | store r r | plus r rn rn |
lea r rn | restrict r r rn |
subseg r rn rn | fail | halt
Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

executionAllowed(\Phi)

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \text{failed} \]
Execution relation

\[ \rightarrow_{\subseteq} (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

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

\[ \text{executionAllowed}(\Phi) \]
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

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

\[ \text{base} \leq a \leq \text{end} \]

\[ \text{executionAllowed}(\Phi) \]
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

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

\[ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}\} \]

\[ \text{executionAllowed} (\Phi) \]
Formalisation - Operational Semantics (1)

**Execution relation**

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

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

\[ base \leq a \leq end \quad perm \in \{\text{rx, rwx}\} \]

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \]

\[ i = \Phi.\text{heap}(a) \]

\[ \Phi \rightarrow J \]

\[ i \rightarrow K \]

\[ \Phi \rightarrow \]
Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

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

\[ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}\} \]

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \text{failed} \]
Formalisation - Operational Semantics (1)

Execution relation

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

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

$$\text{base} \leq a \leq \text{end} \quad perm \in \{\text{rx}, \text{rwx}\}$$

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

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

$$\Phi \rightarrow \text{failed}$$

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

$$\Phi \rightarrow$$
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

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

\[ \text{base} \leq a \leq \text{end} \quad \text{perm} \in \{\text{rx}, \text{rwx}\} \]

\[ \text{executionAllowed}(\Phi) \]

\[ \neg \text{executionAllowed}(\Phi) \]

\[ \Phi \rightarrow \text{failed} \]

\[ \text{executionAllowed}(\Phi) \quad i = \Phi.\text{heap}(a) \]

\[ \Phi \rightarrow i \]
Formalisation - Operational Semantics (1)

Execution relation

\[ \rightarrow \subseteq (\text{Reg} \times \text{Heap}) \times \text{Conf} \]

\[ \Phi.\text{reg}(pc) = (perm, base, end, a) \]
\[ base \leq a \leq end \quad \text{perm} \in \{rx, rwx\} \]
\[ \text{executionAllowed}(\Phi) \quad \neg \text{executionAllowed}(\Phi) \]
\[ \Phi \rightarrow \text{failed} \]

\[ \text{executionAllowed}(\Phi) \quad i = \Phi.\text{heap}(a) \]
\[ \Phi \rightarrow \llbracket i \rrbracket(\Phi) \]
Formalisation - Operational Semantics (2)

\[ [\text{load } r_1 r_2] (\Phi) = \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(r_2) \]

\[ [\text{load } r_1 \ r_2](\Phi) = \Phi[\text{reg.}r_1 \mapsto w] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (perm, base, end, a) \]

\[ [\text{load } r_1 \ r_2](\Phi) = \Phi[\text{reg}.r_1 \mapsto w] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (perm, base, end, a) \]

\[ perm \in \{ro, rw, rx, rwx\} \]

\[ \text{[load } r_1 \ r_2]\](\Phi) = \Phi[\text{reg.}r_1 \mapsto w] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (perm, base, end, a) \]

\[
\begin{align*}
perm & \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \\
\text{base} & \leq a \leq \text{end}
\end{align*}
\]

\[
[\text{load } r_1 \ r_2](\Phi) = \Phi[\text{reg}.r_1 \mapsto w]
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end} \]

\[ [\text{load } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[
\begin{array}{c}
\text{perm} \in \{ \text{ro}, \text{rw}, \text{rx}, \text{rwx} \} \\
\text{base} \leq a \leq \text{end}
\end{array}
\]

\[[\text{load } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{reg.}r_1 \mapsto w])\]

\[
\text{updatePc}(\Phi) = \Phi[\text{reg.}\text{pc} \mapsto \_]
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (perm, base, end, a) \]

\[ perm \in \{ro, rw, rx, rwx\} \quad base \leq a \leq end \]

\[ \text{load } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \]

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

\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.pc \mapsto \ ] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[
\text{perm} \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end}
\]

\[
\text{[load } r_1 \text{ } r_2\text{]}(\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w])
\]

\[
\Phi.\text{reg}(pc) = (\text{perm}, \text{base}, \text{end}, a) \\
\text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \\
\text{updatePc}(\Phi) = \Phi[\text{reg}.pc \mapsto \phantom{]} 
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (perm, base, end, a) \]

\[ perm \in \{ro, rw, rx, rwx\} \quad base \leq a \leq end \]

\[ [\text{load } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \]

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

\[ newPc = (perm, base, end, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.pc \mapsto newPc] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end} \]

\[ [\text{load } r_1 \ r_2](\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \]

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

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

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.\text{pc} \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[
\begin{align*}
\text{perm} & \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \\
\text{base} & \leq a \leq \text{end}
\end{align*}
\]

\[
\text{[load } r_1 \ r_2\text{]}(\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w])
\]

\[
\Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a)
\]

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

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

\[
\text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1)
\]

\[
\text{updatePc}(\Phi) = \Phi[\text{reg}.\text{pc} \mapsto \text{newPc}]
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[
\text{perm} \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end}
\]

\[
\begin{array}{c}
\text{[load } r_1 \ r_2\text{]}(\Phi) = \text{updatePc}(\Phi[\text{reg.}r_1 \mapsto \text{w}]) \\
\end{array}
\]

\[
\Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \\
\text{newPerm} = \text{decodePerm}(\Phi, r_3)
\]

\[
\begin{array}{c}
\text{[restrict } r_1 \ r_2 \ r_3\text{]} = \Phi[\text{reg.}r_1 \mapsto \text{c}] \\
\end{array}
\]

\[
\Phi.\text{reg}(\text{pc}) = (\text{perm}, \text{base}, \text{end}, a) \\
\text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \\
\text{updatePc}(\Phi) = \Phi[\text{reg.}\text{pc} \mapsto \text{newPc}]
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[
\begin{align*}
\text{perm} & \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end} \\
[\text{load } r_1 \ r_2](\Phi) & = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \\
\Phi.\text{reg}(r_2) & = (\text{perm}, \text{base}, \text{end}, a) \\
\text{newPerm} & = \text{decodePerm}(\Phi, r_3) \\
\text{newPerm} & \sqsubseteq \text{perm} \\
[\text{restrict } r_1 \ r_2 \ r_3] & = \Phi[\text{reg}.r_1 \mapsto c] \\
\Phi.\text{reg}(\text{pc}) & = (\text{perm}, \text{base}, \text{end}, a) \\
\text{newPc} & = (\text{perm}, \text{base}, \text{end}, a + 1) \\
\text{updatePc}(\Phi) & = \Phi[\text{reg}.\text{pc} \mapsto \text{newPc}]}
\]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]
\[ \text{perm} \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end} \]
\[ \boxed{\text{load } r_1 \ r_2}(\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \]

\[ \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]
\[ \text{newPerm} = \text{decodePerm}(\Phi, r_3) \]
\[ \text{newPerm} \sqsubseteq \text{perm} \quad c = (\text{newPerm}, \text{base}, \text{end}, a) \]
\[ \boxed{\text{restrict } r_1 \ r_2 \ r_3} = \Phi[\text{reg}.r_1 \mapsto c] \]

\[ \Phi.\text{reg}(\text{pc}) = (\text{perm}, \text{base}, \text{end}, a) \]
\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]
\[ \text{updatePc}(\Phi) = \Phi[\text{reg}.\text{pc} \mapsto \text{newPc}] \]
Formalisation - Operational Semantics (2)

\[ w = \Phi.\text{heap}(a) \quad \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{ro}, \text{rw}, \text{rx}, \text{rwx}\} \quad \text{base} \leq a \leq \text{end} \]

\[ \text{[load } r_1 \ r_2]\] (\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto w]) \]

\[ \Phi.\text{reg}(r_2) = (\text{perm}, \text{base}, \text{end}, a) \]

\[ \text{newPerm} = \text{decodePerm}(\Phi, r_3) \]

\[ \text{newPerm} \sqsubseteq \text{perm} \quad c = (\text{newPerm}, \text{base}, \text{end}, a) \]

\[ \text{[restrict } r_1 \ r_2 \ r_3]\] = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto c]) \]

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

\[ \text{newPc} = (\text{perm}, \text{base}, \text{end}, a + 1) \]

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

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 ()

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.
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.

```ml
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.

```ocaml
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

- 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 during the execution, only interested in the result
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

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 during the execution, only interested in the result
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 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
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. $h(27) \mapsto 5$)
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
- $h : W$
Future World

- Heap changes over time, worlds have to cope with this:
  - $W$
    - Same regions as before
Logical Relation - Worlds, modelling the heap

**Future World**

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

\( W' \)
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
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
Property

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

\[ \mathcal{O} \overset{\text{def}}{=} \lambda W. \{(\text{reg}, h) | \]
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

\[
\mathcal{O} \overset{\text{def}}{=} \lambda W. \{ (\text{reg}, h) \mid (\forall h'. (\text{reg}, h) \rightarrow^* (\text{halted}, h')) \Rightarrow \exists W' \sqsubseteq W. h' : W' \} \lor
\]

\[
(\forall h'. (\text{reg}, h) \rightarrow^* (\text{halted}, h')) \Rightarrow \exists W' \sqsubseteq W. h' : W' \]

\[
(\forall h'. (\text{reg}, h) \rightarrow^* (\text{halted}, h')) \Rightarrow \exists W' \sqsubseteq W. h' : W' \]
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

\[ \mathcal{O} \overset{\text{def}}{=} \lambda W. \{(reg, h) \mid (\forall h'. (reg, h) \rightarrow^* (halted, h')) \Rightarrow \exists W' \sqsupseteq W. h' : W') \lor
\]
\[ (reg, h) \downarrow \lor \]
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

\[ O \overset{\text{def}}{=} \lambda W. \{ (\text{reg}, h) \mid (\forall h'. (\text{reg}, h) \rightarrow^* (\text{halted}, h')) \Rightarrow \exists W' \sqsupseteq W. h' : W' \} \lor \]
\[ (\text{reg}, h) \downarrow \lor \]
\[ (\text{reg}, h) \rightarrow^* \text{failed} \]
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

\[
\mathcal{O} \overset{\text{def}}{=} \lambda W. \{ (\text{reg}, h) \mid (\forall h'. (\text{reg}, h) \rightarrow^* (\text{halted}, h')) \\
\quad \Rightarrow \exists W' \sqsupseteq W. h' : W' \} \lor \\
\quad (\text{reg}, h) \downarrow \lor \\
\quad (\text{reg}, h) \rightarrow^* \text{failed}\}
\]
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

\[ \mathcal{O} \overset{\text{def}}{=} \lambda W. \{ (\text{reg}, h) \mid (\forall h'. (\text{reg}, h) \rightarrow^* (\text{halted}, h')) \Rightarrow \exists W' \sqsubseteq W. h' : W' \} \]
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 ...

\[ E \overset{\text{def}}{=} \lambda W. \{ c \mid \]
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 ...

\[
E \overset{\text{def}}{=} \lambda W. \{ c \mid \forall \text{reg} \in \mathcal{R}(W) \}.
\]
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 ...

\[ \mathcal{E} \overset{\text{def}}{=} \lambda W. \{ c \mid \forall \text{reg} \in \mathcal{R}(W). \forall h : W. \]
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 ...

\[
\mathcal{E} \overset{\text{def}}{=} \lambda W. \{ c \mid \forall \text{reg} \in \mathcal{R}(W). \\
\forall h : W. \\
\quad \forall (\text{reg}[\text{pc} \mapsto c], h) \in \mathcal{O}(W) \}
\]
Logical Relation - Register-file relation

“Well-behaved” register-file

\[ \mathcal{R} \overset{\text{def}}{=} \lambda W. \{ \text{reg} \mid \]
“Well-behaved” register-file

- All registers but the pc-register

\[ \mathcal{R} \overset{\text{def}}{=} \lambda W. \{ \text{reg} \mid \forall r \in \text{RegisterName} \setminus \{\text{pc}\} \}. \]
“Well-behaved” register-file

▷ All registers but the pc-register
  ▷ pc was overwritten in the $E$ anyway

$$R \overset{\text{def}}{=} \lambda W. \{ \text{reg} \mid \forall r \in \text{RegisterName} \setminus \{\text{pc}\} \}. $$
“Well-behaved” register-file

- All registers but the pc-register
  - pc was overwritten in the \( E \) anyway
- should contain a “well-behaved” word

\[
R \overset{\text{def}}{=} \lambda W. \{ \text{reg} \mid \forall r \in \text{RegisterName} \setminus \{ \text{pc} \}. \\
\quad \text{reg}(r) \in \mathcal{V}(W) \}
\]
Logical Relation - Value relation

“Well-behaved words”

\( \mathcal{V} \defeq \lambda W. \{ i \mid i \in \mathbb{Z} \} \cup \{(o, \text{base}, \text{end}, a)\} \cup \{(ro, \text{base}, \text{end}, a)\} \cup \{(rw, \text{base}, \text{end}, a)\} \cup \{(rx, \text{base}, \text{end}, a)\} \cup \{(e, \text{base}, \text{end}, a)\} \cup \{(rwx, \text{base}, \text{end}, a)\} \cup \ldots \)
“Well-behaved words”

\[ \mathcal{V} \overset{\text{def}}{=} \lambda W. \{ i \mid i \in \mathbb{Z}\} \cup \{(o, base, end, a)\} \cup \{(ro, base, end, a) \mid \text{readCondition}(base, end, W)\} \cup \{(rw, base, end, a) \mid \text{readCondition}(base, end, W) \wedge \} \cup \{(rx, base, end, a) \mid \text{readCondition}(base, end, W) \wedge \} \cup \{(e, base, end, a) \mid \}\cup \{(rwx, base, end, a) \mid \text{readCondition}(base, end, W) \wedge \wedge \}\]
Logical Relation - Value relation

“Well-behaved words”

\[ \mathcal{V} \overset{\text{def}}{=} \lambda \ W. \ \{ i \ | \ i \in \mathbb{Z} \} \cup \]
\[ \quad \{(0, \ \text{base}, \ \text{end}, \ a)\} \cup \]
\[ \quad \{(r_0, \ \text{base}, \ \text{end}, \ a) \mid \text{readCondition}(\text{base}, \ \text{end}, \ W)\} \cup \]
\[ \quad \{(r_w, \ \text{base}, \ \text{end}, \ a) \mid \text{readCondition}(\text{base}, \ \text{end}, \ W) \land \]
\[ \qquad \text{writeCondition}(\text{base}, \ \text{end}, \ W)\} \cup \]
\[ \quad \{(r_x, \ \text{base}, \ \text{end}, \ a) \mid \text{readCondition}(\text{base}, \ \text{end}, \ W) \land \]
\[ \qquad \} \cup \]
\[ \quad \{(e, \ \text{base}, \ \text{end}, \ a) \mid \]
\[ \quad \{(rwx, \ \text{base}, \ \text{end}, \ a) \mid \text{readCondition}(\text{base}, \ \text{end}, \ W) \land \]
\[ \qquad \text{writeCondition}(\text{base}, \ \text{end}, \ W)\} \land \]
Logical Relation - Value relation

“Well-behaved words”

\[ \mathcal{V} \overset{\text{def}}{=} \lambda W. \{ i \mid i \in \mathbb{Z} \} \cup \]
\[ \{ (o, base, end, a) \} \cup \]
\[ \{ (ro, base, end, a) \mid \text{readCondition}(base, end, W) \} \cup \]
\[ \{ (rw, base, end, a) \mid \text{readCondition}(base, end, W) \land \]
\[ \text{writeCondition}(base, end, W) \} \cup \]
\[ \{ (rx, base, end, a) \mid \text{readCondition}(base, end, W) \land \]
\[ \text{executeCondition}(base, end, W) \} \cup \]
\[ \{ (e, base, end, a) \mid \} \cup \]
\[ \{ (rwx, base, end, a) \mid \text{readCondition}(base, end, W) \land \]
\[ \text{writeCondition}(base, end, W) \land \]
\[ \text{executeCondition}(base, end, W) \} \]
Logical Relation - Value relation

“Well-behaved words”

\[ \mathcal{V} \overset{\text{def}}{=} \lambda W. \{ i \mid i \in \mathbb{Z} \} \cup \{(o, \text{base}, \text{end}, a)\} \cup \{(r_0, \text{base}, \text{end}, a) \mid \text{readCondition}(\text{base}, \text{end}, W)\} \cup \{(r_w, \text{base}, \text{end}, a) \mid \text{readCondition}(\text{base}, \text{end}, W) \land \text{writeCondition}(\text{base}, \text{end}, W)\} \cup \{(r_x, \text{base}, \text{end}, a) \mid \text{readCondition}(\text{base}, \text{end}, W) \land \text{executeCondition}(\text{base}, \text{end}, W)\} \cup \{(e, \text{base}, \text{end}, a) \mid \text{enterCondition}(\text{base}, \text{end}, a, W)\} \cup \{(r_{wx}, \text{base}, \text{end}, a) \mid \text{readCondition}(\text{base}, \text{end}, W) \land \text{writeCondition}(\text{base}, \text{end}, W) \land \text{executeCondition}(\text{base}, \text{end}, W)\} \]
Logical Relation - Execute and enter conditions

**Execution condition**

\[ \text{executeCondition}(\text{base}, \text{end}, W) \overset{\text{def}}{=} \]

\[ \text{...} \]
**Execution condition**

- May be used at any point in the future

\[
\text{executeCondition}(base, end, W) \overset{\text{def}}{=} \forall W' \sqsubseteq W.
\]
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

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

$$\forall W' \sqsubseteq W.$$

$$\forall a \in [\text{base}, \text{end}].$$
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 $E$-relation

$$\text{executeCondition}(\text{base}, \text{end}, W) \overset{def}{=} \forall W' \sqsubseteq W. \forall a \in [\text{base}, \text{end}]. (rx, \text{base}, \text{end}, a) \in E(W')$$
Logical Relation - Execute and enter conditions

Enter condition

\[ enterCondition(base, end, a, W) \overset{\text{def}}{=} \]
Enter condition
- May be used at any point in the future

\[ enterCondition(base, end, a, W) \overset{\text{def}}{=} \forall W' \sqsupseteq W. \]
Enter condition

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

\[
\text{enterCondition}(\text{base}, \text{end}, a, W) \overset{\text{def}}{=} \forall W' \sqsubseteq W.
\]
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 $E$-relation

$$
\text{enterCondition}(\text{base, end, } a, W) \overset{\text{def}}{=} \\
\forall W' \supseteq W. \\
(rx, \text{base, end, } a) \in E(W')
$$
Logical Relation - Read and write conditions

Read condition

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

\[
\text{readCondition}(base, end, W) \overset{\text{def}}{=} \]

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

\[
\text{readCondition}(\text{base}, \text{end}, W) \overset{\text{def}}{=} \exists r \in \text{RegionName}.
\]
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

\[
\text{readCondition}(\text{base}, \text{end}, W) \overset{\text{def}}{=} \\
\exists r \in \text{RegionName}. \\
\exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}].
\]
Logical Relation - Read and write conditions

Read condition

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

\[
\text{readCondition}(base, end, W) \overset{\text{def}}{=} \\
\exists r \in \text{RegionName}. \\
\exists [base', end'] \supseteq [base, end]. \\
W(r) \subseteq \iota_{base',end'}
\]
Logical Relation - Read and write conditions

Read condition

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

$\text{readCondition}(base, end, W) \overset{\text{def}}{=} \exists r \in \text{RegionName}. \exists [base', end'] \supseteq [base, end]. W(r) \subseteq \nu_{base', end'}$

- $\nu_{base', end'}$ is a standard region that requires
Logical Relation - Read and write conditions

Read condition

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

\[
\text{readCondition}(base, \text{end}, W) \overset{\text{def}}{=} \\
\exists r \in \text{RegionName}. \\
\exists [base', end'] \supseteq [base, end]. \\
W(r) \subseteq \iota_{base',end'}
\]

- $\iota_{base',end'}$ 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 $\iota_{base', end'}$

\[
\text{readCondition}(base, end, W) \overset{\text{def}}{=} \\
\exists r \in \text{RegionName}. \\
\exists [base', end'] \supseteq [base, end]. \\
W(r) \subseteq \iota_{base', end'}
\]

- $\iota_{base', end'}$ 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 $\nu$-relation
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.

$$\text{readCondition}(base, end, W) \overset{\text{def}}{=} \exists r \in \text{RegionName}.
\quad \exists [base', end'] \supseteq [base, end].
\quad W(r) \subseteq \iota_{base',end'}$$

- $\iota_{base',end'}$ 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 to write

\[
\text{writeCondition}(\text{base}, \text{end}, W) \overset{\text{def}}{=} \]

- \( \iota_{\text{base}', \text{end}'} \) is a standard region that requires
  - Range of heap segment to be \([\text{base}', \text{end}']\)
  - All the words in the heap segment should be in the \( \forall \)-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

\[ \text{writeCondition}(\text{base}, \text{end}, W) \overset{\text{def}}{=} \]

\[ \exists r \in \text{RegionName}. \]

\[ \exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \]

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

\[
\text{writeCondition}(\text{base}, \text{end}, W) \overset{\text{def}}{=} \\
\exists r \in \text{RegionName}. \\
\exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \\
W(r) \supseteq \iota_{\text{base}',\text{end}'}
\]

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

$$writeCondition(base, end, W) \overset{def}{=} \exists r \in RegionName. \exists [base', end'] \supseteq [base, end]. \ W(r) \supseteq \iota_{base', end'}$$

- $\iota_{base', end'}$ 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 $\forall$-relation
Lemma (FTLR)

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

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

For all \( W \in \text{World} \), \( perm \in \text{Perm} \), and \( base, end, a \in \text{Addr} \),

if

\[ perm = \text{rx} \text{ and } \text{readCondition}(W, base, end), \]

or

\[ perm = \text{rwx} \text{ and } \text{read-/writeCond}(W, base, end) \]

then

\[ (perm, base, end, a) \in \mathcal{E}(W). \]
Road map

Capability Machine

Formalisation

Example program

Logical Relation

Example revisited

Current work
Example: local state revisited

Lemma
Given any program \( \text{adv} \), \( f(\text{adv}) \) either runs forever, ends up in the failed configuration, or halts in a configuration where the assertion flag is 0.

\[
\text{let } f = \text{ fun } \text{adv} => \\
\text{ let } l = 1 \text{ in } \\
\text{ adv(); } \\
\text{ assert } (l == 1)
\]

Example: local state revisited

Proof sketch

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

```ocaml
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`

```latex
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:

```ocaml
let f = fun adv =>
    let l = 1 in
    adv();
    assert (l == 1)
```
Example: local state revisited

Proof sketch

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

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

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

```ocaml
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
  - 1 remains 1
  - standard region governs adv

```ml
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 $f = \text{fun } adv \Rightarrow$
  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

```haskell
let f = fun adv =>
    let l = 1 in
    adv();
    assert (l == 1)
```
let f = fun adv =>
    let l = 1 in
    adv();
    assert (l == 1)
Example: local state revisited

Proof sketch (continued)

- Use FTLR on \( \text{adv} \) capability
- By design, the heap satisfies the world

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

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

```ocaml
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 $R$-relation:
  - All registers but two contain 0, so trivial.
  - One is pc-register, so we don’t care about it.

```ocaml
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 $R$-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

```ocaml
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
  - 1 remains 1
  - assertion flag is 0
- The continuation satisfies enterCondition:

```ocaml
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
  - 1 remains 1
  - assertion flag is 0

- The continuation satisfies `enterCondition`:
  - In a future world, the continuation must be in $E$

```ml
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 \textit{enterCondition}:
  ▶ In a future world, the continuation must be in $\mathcal{E}$
  ▶ Executing from continuation, l is still 1, so assertion does not fail.

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

```ocaml
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 $R$-relation

```ocaml
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 $\text{adv} \in \mathcal{E}$: execution diverges, fails, or terminates without the assertion failing.

```ocaml
let f = fun adv =>
    let l = 1 in
    adv();
    assert (l == 1)
```
Example: local state revisited

Lemma
Given any program $\text{adv}$, $f(\text{adv})$ either runs forever, ends up in the failed configuration, or halts in a configuration where the assertion flag is 0.

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

Capability Machine

Formalisation

Example program

Logical Relation

Example revisited

Current work
What are we working on now?

```plaintext
let f = fun adv =>
    let l = 0 in
    adv();
    assert(l == 0);
    l := 1;
    adv()
```
What are we working on now?

```latex
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?

```ocaml
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?

```ocaml
define 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?

```plaintext
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?

```ocaml
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?

```ml
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?