Defining contextual refinement for capability machines

Dorian Lesbre

March 13th, 2023

Logic and Semantics Seminar

Outline

1. Introduction
2. Capability machines
3. Components and contexts
4. Defining contextual refinement
5. Validity relation
6. Conclusion

Introduction

Contextual refinement

- Binary relation between two open programs
- Any observable behavior from p is also observable in p'

Contextual refinement

- Binary relation between two open programs
- Any observable behavior from p is also observable in p'

General definition

$$p \preccurlyeq_c p' \quad := \quad \forall\, C, C[p] \text{ terminates } \Rightarrow \; C[p'] \text{ terminates}$$

Applications of contextual refinement

- Reasoning on open programs using the concrete semantics

Applications of contextual refinement

- Reasoning on open programs using the concrete semantics
- Specify a program in terms of another

Applications of contextual refinement

- Reasoning on open programs using the concrete semantics
- Specify a program in terms of another
- Express representation independence

Applications of contextual refinement

- Reasoning on open programs using the concrete semantics
- Specify a program in terms of another
- Express representation independence
- Reasoning algebraically about program constructs

Applications of contextual refinement

- Reasoning on open programs using the concrete semantics
- Specify a program in terms of another
- Express representation independence
- Reasoning algebraically about program constructs

BUT: often hard to prove

Example: specification as a program

Formal specification:

$$
\begin{array}{l}
\forall\, P, I, f, xs, \ell, \\
\quad \left\{ \begin{array}{l} \texttt{isList}\ \ell\ xs * \texttt{all}\ P\ xs * I\ []\ a\ * \\ (\forall x,\ x,\ a',\ ys, \{P\ x * I\ ys\ a'\}\ \ f\ x\ a'\ \ \{r.\ I\ (x :: ys)\ r\}) \end{array} \right\} \\
\quad \texttt{foo}\ f\ a\ \ell \\
\quad \{r.\ \texttt{isList}\ \ell\ xs * I\ xs\ r\}
\end{array}
$$

Example: specification as a program

Formal specification:

$$\forall\, P,\, I,\, f,\, xs,\, \ell,$$
$$\left\{ \begin{array}{l} \texttt{isList}\ \ell\ xs * \texttt{all}\ P\ xs * I\ []\ a\ * \\ (\forall x,\ x,\ a',\ ys, \{P\ x * I\ ys\ a'\}\ f\ x\ a'\ \{r.\ I\ (x :: ys)\ r\}) \end{array} \right\}$$
$$\texttt{foo}\ f\ a\ \ell$$
$$\{r.\ \texttt{isList}\ \ell\ xs * I\ xs\ r\}$$

Specification as a program:

```
let rec foo_spec f a l = match l with
  | [] -> a
  | x::xs -> f x (foo_spec f a xs)
```

Example: representation independence

```
let counter () =
  let x = ref 0 in
  let incr () =
    x := !x + 1
  in
  let read () = !x
  in incr, read
```

```
let counter_neg () =
  let x = ref 0 in
  let incr () =
    x := !x - 1
  in
  let read () = - !x
  in incr, read
```

Capability machines

What is a capability machine

- Security oriented CPU
- Check memory access via special machine words:

$$\mathtt{Word} = \mathbb{Z} \sqcup \mathtt{Cap}$$

What is a capability machine

- Security oriented CPU
- Check memory access via special machine words:

$$\texttt{Word} = \mathbb{Z} \sqcup \texttt{Cap}$$

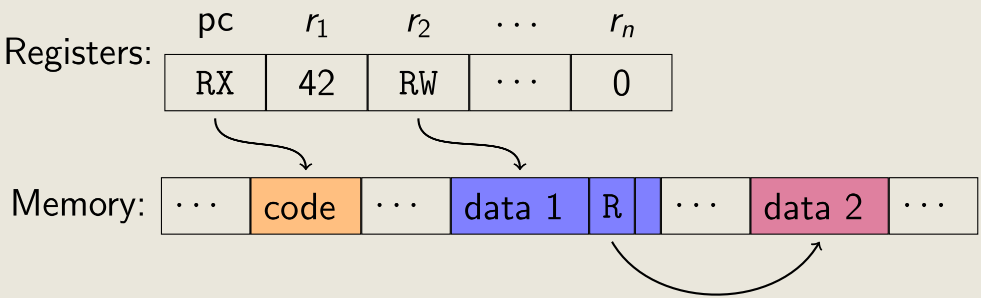
Capability

$$c \in \texttt{Cap} \;:=\; (p, b, e, a)$$

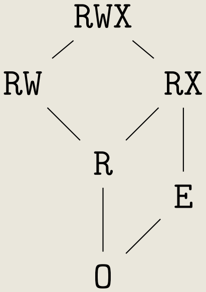
where $p \in \{\texttt{O},\ \texttt{E},\ \texttt{R},\ \texttt{RW},\ \texttt{RX},\ \texttt{RWX}\}$

$\Rightarrow$ gives access to $[b; e)$ with permission p

Memory access via capabilities



Permission order



Memory access via capabilities

Very few instructions modify capabilities :

- `lea` r z changes a capability's address to $a + z$

Memory access via capabilities

Very few instructions modify capabilities :

- `lea` r z changes a capability's address to $a + z$
- `subseg` r b' e' modifies the range to $[b'; e') \subseteq [b; e)$

Memory access via capabilities

Very few instructions modify capabilities :

- `lea` r z changes a capability's address to $a + z$
- `subseg` r b' e' modifies the range to $[b'; e') \subseteq [b; e)$
- `restrict` r p' modifies the permission to $p' \preccurlyeq p$

Memory access via capabilities

Very few instructions modify capabilities :

- $\texttt{lea}\ r\ z$ changes a capability's address to $a + z$
- $\texttt{subseg}\ r\ b'\ e'$ modifies the range to $[b'; e') \subseteq [b; e)$
- $\texttt{restrict}\ r\ p'$ modifies the permission to $p' \preccurlyeq p$
- $\texttt{jmp}\ r$ and $\texttt{jnz}\ r\ \rho$ change $\texttt{E}$ to $\texttt{RX}$.

Cerise capability machine model

Simple model:

- Single core
- No interruptions
- No privilege levels
- No virtual memory
- Limited instruction set

Cerise capability machine model

Simple model:

- Single core
- No interruptions
- No privilege levels
- No virtual memory
- Limited instruction set

But captures:

- Finite memory
- Fixed set of registers
- Instructions encoded as integers

Cerise instruction set

$$\rho \in \mathbb{Z} \sqcup \mathtt{RegName}$$

$$\begin{aligned} i \in \mathtt{Instr} \quad := \quad & \mathtt{fail} \mid \mathtt{halt} \mid \mathtt{jmp}\ r \mid \mathtt{jnz}\ r\ r \mid \\ & \mathtt{move}\ r\ \rho \mid \mathtt{load}\ r\ r \mid \mathtt{store}\ r\ \rho \mid \\ & \mathtt{add}\ r\ \rho\ \rho \mid \mathtt{sub}\ r\ \rho\ \rho \mid \mathtt{lt}\ r\ \rho\ \rho \mid \\ & \mathtt{restrict}\ r\ \rho \mid \mathtt{subseg}\ r\ \rho\ \rho \mid \mathtt{lea}\ r\ \rho \mid \mathtt{isptr}\ r\ r \mid \\ & \mathtt{getp}\ r\ r \mid \mathtt{getb}\ r\ r \mid \mathtt{gete}\ r\ r \mid \mathtt{geta}\ r\ r \end{aligned}$$

Machine state

$$
\begin{aligned}
(\mathtt{mem}, \mathtt{regs}) \in \mathtt{ExecConf} \quad &:= \quad (\mathtt{Addr} \to \mathtt{Word}) \times (\mathtt{RegName} \to \mathtt{Word}) \\
\delta \in \mathtt{ExecMode} \quad &:= \quad \mathtt{Halted} \mid \mathtt{Failed} \mid \mathtt{Running}
\end{aligned}
$$

Machine state: $\mathtt{ExecMode} \times \mathtt{ExecConf}$

Small step semantics

$\textsc{ExecStep}$

$$(\mathtt{Running}, (\mathtt{mem}, \mathtt{regs})) \rightarrow \begin{cases} \mathtt{execInstr}\ \mathtt{mem}\ \mathtt{regs}\ i & \text{if } \mathtt{regs}(\mathtt{pc}) = (p, b, e, a) \wedge \\ & \mathtt{RX} \preccurlyeq p \wedge a \in [b; e) \wedge \\ & \mathtt{decodeInstr}(\mathtt{mem}(a)) = \mathtt{Some}\ i \\ \mathtt{Failed}, (\mathtt{mem}, \mathtt{regs}) & \text{otherwise} \end{cases}$$

Components and contexts

Defining open and closed program

What is a program?

Defining open and closed program

What is a program?

- A region of memory containing encoded instructions
- A register state $\texttt{RegName} \to \texttt{Word}$

Defining open and closed program

What is a program?

- A region of memory containing encoded instructions
- A register state $\texttt{RegName} \to \texttt{Word}$

An open program?

A closed program?

A context?

Defining open programs: components

Open program:

- segment of memory
- interface to access it

Defining open programs: components

Open program:

- segment of memory
- interface to access it

Component

$$\texttt{component} \;:=\; \left\{ \begin{array}{lcl} \texttt{segment} & : & \texttt{Addr} \rightharpoonup \texttt{Word} \\ \texttt{imports} & : & \texttt{Addr} \rightharpoonup \texttt{Symbols} \\ \texttt{exports} & : & \texttt{Symbols} \rightharpoonup \texttt{Word} \end{array} \right\}$$

Well-formed components

- imports and exports symbols are disjoint:
  $\mathtt{img}\,(\mathtt{imports}) \cap \mathtt{dom}\,(\mathtt{exports}) = \emptyset$

Well-formed components

- imports and exports symbols are disjoint:
  $\mathtt{img}\,(\mathtt{imports}) \cap \mathtt{dom}\,(\mathtt{exports}) = \emptyset$
- import addresses are part of the component's memory:
  $\mathtt{dom}\,(\mathtt{imports}) \subseteq \mathtt{dom}\,(\mathtt{segment})$

Well-formed components

- imports and exports symbols are disjoint:
  $\mathtt{img}\,(\mathtt{imports}) \cap \mathtt{dom}\,(\mathtt{exports}) = \emptyset$
- import addresses are part of the component's memory:
  $\mathtt{dom}\,(\mathtt{imports}) \subseteq \mathtt{dom}\,(\mathtt{segment})$
- contained capabilities only point to its memory:

$$\forall\,(\_, b, e, \_) \in \mathtt{img\ segment} \cup \mathtt{img\ exports},$$
$$[b; e) \subseteq \mathtt{dom\ segment}$$

Closed program

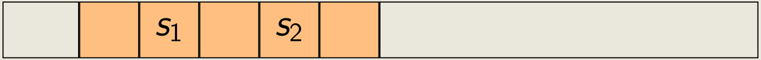
Program

A program is a pair $(p, \texttt{regs})$:

- p is a well-formed component with no imports
- $\texttt{regs} \in \texttt{RegName} \to \texttt{Word}$ is a register state
- capabilities in $\texttt{regs}$ point to p

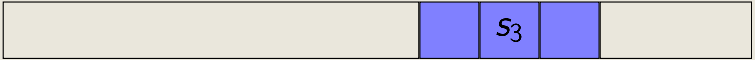
Linking

x:



$$\texttt{exports} = \{s_3 \mapsto w_3,\ s_4 \mapsto w_4\}$$

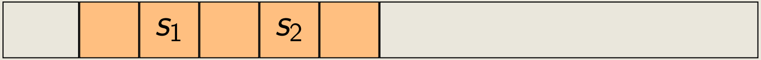
y:



$$\texttt{exports} = \{s_1 \mapsto w_1\}$$

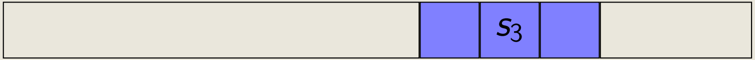
Linking

x:



$\texttt{exports} = \{s_3 \mapsto w_3,\ s_4 \mapsto w_4\}$

y:



$\texttt{exports} = \{s_1 \mapsto w_1\}$

$x \bowtie y$:

$\texttt{exports} = \{s_3 \mapsto w_3,\ s_4 \mapsto w_4,\ s_1 \mapsto w_1\}$

Linking

Requires components to be **disjoint** and well-formed:

$$x \bowtie y :=$$

$$\left\{ \begin{array}{lcl} \texttt{exports} & := & x.\texttt{exports} \uplus y.\texttt{exports} \\ \texttt{imports} & := & \left\{ a \mapsto s \;\middle|\; \begin{array}{l} a \mapsto s \in x.\texttt{imports} \uplus y.\texttt{imports} \wedge \\ s \mapsto \_ \notin x.\texttt{exports} \uplus y.\texttt{exports} \end{array} \right\} \\ \texttt{segment} & := & x.\texttt{segment}[y.\texttt{exports} \circ x.\texttt{imports}] \uplus \\ & & y.\texttt{segment}[x.\texttt{exports} \circ y.\texttt{imports}] \end{array} \right\}$$

Properties of linking

- $x \#_\ell y \Rightarrow x \bowtie y$ well-formed
- commutative: $x \#_\ell y \Rightarrow x \bowtie y = y \bowtie x$
- associative:
  $x \#_\ell y \wedge y \#_\ell z \wedge x \#_\ell z \Rightarrow x \bowtie (y \bowtie z) = (x \bowtie y) \bowtie z$

Context

"Just what is needed" to turn a component into a program.

Context

"Just what is needed" to turn a component into a program.

Context

A context for a component x is a pair $(z, \mathtt{regs})$ where:

- $x \#_\ell z$
- $\mathtt{img}\ x.\mathtt{imports} \subseteq \mathtt{dom}\ z.\mathtt{exports}$
- $\mathtt{img}\ z.\mathtt{imports} \subseteq \mathtt{dom}\ x.\mathtt{exports}$
- capabilities in $\mathtt{regs}$ point to z

Properties of context

- $(z, \mathtt{regs})$ is a context of $x \Rightarrow (z \bowtie x, \mathtt{regs})$ is a program

Properties of context

- $(z, \mathtt{regs})$ is a context of $x \Rightarrow (z \bowtie x, \mathtt{regs})$ is a program
- $(z, \mathtt{regs})$ is a context of $x \bowtie y \Leftrightarrow$
  $(z \bowtie x, \mathtt{regs})$ is a context of y and
  capabilities in $\mathtt{regs}$ point to z

Properties of context

- $(z, \texttt{regs})$ is a context of $x \Rightarrow (z \bowtie x, \texttt{regs})$ is a program
- $(z, \texttt{regs})$ is a context of $x \bowtie y \Leftrightarrow$
  $(z \bowtie x, \texttt{regs})$ is a context of y and
  capabilities in $\texttt{regs}$ point to z
- if $y.\texttt{exports} = \emptyset$ and $(z, \texttt{regs})$ is a context of $x \bowtie y$ then
  $(z, \texttt{regs})$ is a context of y

Defining contextual refinement

Contextual refinement

General idea: $x \preccurlyeq_{\text{ctx}} y$ when:

- for all context $(z, \texttt{regs})$
- for all values $v \in \{\texttt{Halted}, \texttt{Failed}\}$

if $\exists n,\ \texttt{machine\_run}\ n\ (z \bowtie x)\ \texttt{regs} = v$
then $\exists n,\ \texttt{machine\_run}\ n\ (z \bowtie y)\ \texttt{regs} = v$

Contextual refinement

General idea: $x \preccurlyeq_{\text{ctx}} y$ when:

- **for all context** $(z, \texttt{regs})$
- for all values $v \in \{\texttt{Halted}, \texttt{Failed}\}$

if $\exists n,\ \texttt{machine\_run}\ n\ (z \bowtie x)\ \texttt{regs} = v$

then $\exists n,\ \texttt{machine\_run}\ n\ (z \bowtie y)\ \texttt{regs} = v$

What quantification on context?

Multiple options:

1. quantify on context of both x and y

What quantification on context?

Multiple options:

1. quantify on context of both x and y
   - allows x and y to be VERY different
   - weak refinement, no transitivity

What quantification on context?

Multiple options:

1. quantify on context of both x and y
   - allows x and y to be VERY different
   - weak refinement, no transitivity
2. require that all contexts of x be contexts of y

What quantification on context?

Multiple options:

1. quantify on context of both x and y
   - allows x and y to be VERY different
   - weak refinement, no transitivity
2. require that all contexts of x be contexts of y
   - unpleasant side conditions ($y \subseteq x$)
   - stronger refinement

What quantification on context?

Multiple options:

1. quantify on context of both x and y
   - allows x and y to be VERY different
   - weak refinement, no transitivity
2. require that all contexts of x be contexts of y
   - unpleasant side conditions ($y \subseteq x$)
   - stronger refinement
3. define a type system on components

What quantification on context?

Multiple options:

1. quantify on context of both x and y
   - allows x and y to be VERY different
   - weak refinement, no transitivity
2. require that all contexts of x be contexts of y
   - unpleasant side conditions ($y \subseteq x$)
   - stronger refinement
3. define a type system on components
   - complex type system
   - strongest refinement, implies point 2

What quantification on context?

Multiple options:

1. quantify on context of both x and y
   - allows x and y to be VERY different
   - weak refinement, no transitivity
2. require that all contexts of x be contexts of y
   - unpleasant side conditions ($y \subseteq x$)
   - stronger refinement
3. define a type system on components
   - complex type system
   - strongest refinement, implies point 2

Contextual refinement

Improved definition: $x \preccurlyeq_{\text{ctx}} y$ when:

- for all $(z, \texttt{regs})$
- for all values $v \in \{\texttt{Halted}, \texttt{Failed}\}$

$$\begin{cases} (z, \texttt{regs}) \text{ is a context of } x \\ \exists n,\ \texttt{machine\_run}\ n\ (z \bowtie x)\ \texttt{regs} = v \end{cases} \Rightarrow$$

$$\begin{cases} (z, \texttt{regs}) \text{ is a context of } y \\ \exists n,\ \texttt{machine\_run}\ n\ (z \bowtie y)\ \texttt{regs} = v \end{cases}$$

Contextual refinement

Improved definition: $x \preccurlyeq_{\text{ctx}} y$ when:

- for all $(z, \texttt{regs})$
- for all values $v \in \{\texttt{Halted}, \texttt{Failed}\}$

$$\begin{cases} (z, \texttt{regs}) \text{ is a context of } x \\ \exists n,\ \texttt{machine\_run}\ n\ (z \bowtie x)\ \texttt{regs} = v \end{cases} \Rightarrow$$

$$\begin{cases} (z, \texttt{regs}) \text{ is a context of } y \\ \exists n,\ \texttt{machine\_run}\ n\ (z \bowtie y)\ \texttt{regs} = v \end{cases}$$

A few problems remain

Empty quantification: because of finite memory

A few problems remain

Empty quantification: because of finite memory

$\Rightarrow$ require that x leave some space free

$\Rightarrow$ helps with proofs as well

A few problems remain

Empty quantification: because of finite memory

$\Rightarrow$ require that x leave some space free

$\Rightarrow$ helps with proofs as well

Components can be too different:

A few problems remain

Empty quantification: because of finite memory

$\Rightarrow$ require that x leave some space free

$\Rightarrow$ helps with proofs as well

Components can be too different:

$\Rightarrow$ require that $\texttt{dom}\ y\texttt{.exports} \subseteq \texttt{dom}\ x\texttt{.exports}$

Final definition

- $\text{dom } x.\mathit{segment} \cap [0; \text{ctxt\_size}) = \emptyset$
- $\text{dom } y.\texttt{exports} \subseteq \text{dom } x.\texttt{exports}$
- for all $(z, \texttt{regs})$, for all $v \in \{\texttt{Halted}, \texttt{Failed}\}$

$$\left\{\begin{array}{l} (z, \texttt{regs}) \text{ is a context of } x \\ \exists n,\ \texttt{machine\_run}\ n\ (z \bowtie x)\ \texttt{regs} = v \end{array}\right. \Rightarrow$$

$$\left\{\begin{array}{l} (z, \texttt{regs}) \text{ is a context of } y \\ \exists n,\ \texttt{machine\_run}\ n\ (z \bowtie y)\ \texttt{regs} = v \end{array}\right.$$

Good properties of contextual refinement

non-trivial: $\exists\, x\, y,\ x \neq y \ \wedge\ x \ \preccurlyeq_{\mathsf{ctx}}\ y$

Good properties of contextual refinement

non-trivial: $\exists\, x\, y,\ x \neq y \ \wedge\ x \ \preccurlyeq_{\text{ctx}}\ y$

reflexive: x well-formed $\ \Rightarrow\ x \ \preccurlyeq_{\text{ctx}}\ x$

Good properties of contextual refinement

non-trivial: $\exists\, x\, y,\ x \neq y \ \wedge\ x \ \preccurlyeq_{\text{ctx}}\ y$

reflexive: x well-formed $\ \Rightarrow\ x \ \preccurlyeq_{\text{ctx}}\ x$

transitive: $x \ \preccurlyeq_{\text{ctx}}\ y \wedge y \ \preccurlyeq_{\text{ctx}}\ z \ \Rightarrow\ x \ \preccurlyeq_{\text{ctx}}\ z$

Good properties of contextual refinement

non-trivial: $\exists\, x\, y,\ x \neq y \ \wedge\ x \ \preccurlyeq_{\text{ctx}}\ y$

reflexive: x well-formed $\ \Rightarrow\ x \ \preccurlyeq_{\text{ctx}}\ x$

transitive: $x \ \preccurlyeq_{\text{ctx}}\ y \wedge y \ \preccurlyeq_{\text{ctx}}\ z \ \Rightarrow\ x \ \preccurlyeq_{\text{ctx}}\ z$

compositional: if x and y disjoint

$$x \ \preccurlyeq_{\text{ctx}}\ x' \wedge y \ \preccurlyeq_{\text{ctx}}\ y' \ \Rightarrow\ (x \ \bowtie\ y) \ \preccurlyeq_{\text{ctx}}\ (x' \ \bowtie\ y')$$

Other properties of contextual refinement

Other consequences: if $x \preccurlyeq_{\text{ctx}} y$ then

- All public memory of x and y is the same

Other properties of contextual refinement

Other consequences: if $x \preccurlyeq_{\mathrm{ctx}} y$ then

- All public memory of x and y is the same
- Depends on absolute memory position

Other properties of contextual refinement

Other consequences: if $x \preccurlyeq_{\mathrm{ctx}} y$ then

- All public memory of x and y is the same
- Depends on absolute memory position
- Non-terminating programs refine pretty-much anything

Other properties of contextual refinement

Other consequences: if $x \preccurlyeq_{\mathrm{ctx}} y$ then

- All public memory of x and y is the same
- Depends on absolute memory position
- Non-terminating programs refine pretty-much anything
- E capabilities behave in the same way

Other properties of contextual refinement

Other consequences: if $x \preccurlyeq_{\mathrm{ctx}} y$ then

- All public memory of x and y is the same
- Depends on absolute memory position
- Non-terminating programs refine pretty-much anything
- E capabilities behave in the same way
- $\mathrm{dom}(\texttt{segment y}) \subseteq \mathrm{dom}(\texttt{segment x})$

Growing and shrinking components

if z has no exports then:

- if $x \preccurlyeq_{\text{ctx}} y$ then $x \bowtie z \preccurlyeq_{\text{ctx}} y$
- if $x \preccurlyeq_{\text{ctx}} y \bowtie z$ then $x \preccurlyeq_{\text{ctx}} y$

Validity relation

Unary validity relation

Goal: capture values safe to share with unknown code

$$
\begin{aligned}
\mathcal{V}(z) &:= \texttt{True} \\
\mathcal{V}(\texttt{0}, b, e, a) &:= \texttt{True}
\end{aligned}
$$

Unary validity relation

Goal: capture values safe to share with unknown code

$$\begin{aligned}
\mathcal{V}(z) &:= \texttt{True} \\
\mathcal{V}(\texttt{O}, b, e, a) &:= \texttt{True} \\
\mathcal{V}(\texttt{E}, b, e, a) &:= \triangleright \Box\, \mathcal{E}(\texttt{RX}, b, e, a)
\end{aligned}$$

Unary validity relation

Goal: capture values safe to share with unknown code

$$
\begin{aligned}
\mathcal{V}(z) &:= \texttt{True} \\
\mathcal{V}(\texttt{O}, b, e, a) &:= \texttt{True} \\
\mathcal{V}(\texttt{E}, b, e, a) &:= \triangleright \Box\, \mathcal{E}(\texttt{RX}, b, e, a) \\
\mathcal{V}(\texttt{R}/\texttt{RX}, b, e, a) &:= \mathop{\Huge *}_{a \in [b;e)} \exists P, \begin{cases} \boxed{\exists w,\ a \mapsto_a w \ * \ P(w)} \ * \\ \triangleright \Box\, \forall w,\ P(w) \ -\!\!* \ \mathcal{V}(w) \end{cases}
\end{aligned}
$$

Unary validity relation

Goal: capture values safe to share with unknown code

$$
\begin{aligned}
\mathcal{V}(z) &:= \texttt{True} \\
\mathcal{V}(\texttt{0}, b, e, a) &:= \texttt{True} \\
\mathcal{V}(\texttt{E}, b, e, a) &:= \triangleright \Box\, \mathcal{E}(\texttt{RX}, b, e, a) \\
\mathcal{V}(\texttt{R}/\texttt{RX}, b, e, a) &:= \mathop{\Huge\ast}_{a \in [b;e)} \exists P, \begin{cases} \boxed{\exists w,\ a \mapsto_a w \ * \ P(w)} \ * \\ \triangleright \Box\, \forall w,\ P(w) \ \mathrel{-\!\!*}\ \mathcal{V}(w) \end{cases} \\
\mathcal{V}(\texttt{RW}/\texttt{RWX}, b, e, a) &:= \mathop{\Huge\ast}_{a \in [b;e)} \boxed{\exists w,\ a \mapsto_a w \ * \ \mathcal{V}(w)}
\end{aligned}
$$

Unary validity relation

Goal: capture values safe to share with unknown code

$$
\begin{array}{lcl}
\mathcal{V}(z) & := & \mathtt{True} \\
\mathcal{V}(\mathtt{O}, b, e, a) & := & \mathtt{True} \\
\mathcal{V}(\mathtt{E}, b, e, a) & := & \triangleright \Box\, \mathcal{E}(\mathtt{RX}, b, e, a) \\
\mathcal{V}(\mathtt{R}/\mathtt{RX}, b, e, a) & := & \mathop{\Huge *}\limits_{a \in [b;e)} \exists\, P,\ \left\{ \begin{array}{l} \boxed{\exists\, w,\ a \mapsto_a w \ * \ P(w)} \ * \\ \triangleright \Box\, \forall\, w,\ P(w) \ -\!\!* \ \mathcal{V}(w) \end{array} \right. \\
\mathcal{V}(\mathtt{RW}/\mathtt{RWX}, b, e, a) & := & \mathop{\Huge *}\limits_{a \in [b;e)} \boxed{\exists\, w,\ a \mapsto_a w \ * \ \mathcal{V}(w)}
\end{array}
$$

Recursive definition possible thanks to Iris' later modality ($\triangleright$)

Unary expression relation

Goal: capture values safe to execute with unknown code

$$
\begin{aligned}
\mathcal{E}(w) \;:=\; & \forall\, \mathtt{regs} \in \mathtt{RegName} \to \mathtt{Addr},\; \mathtt{regs}_{\ell}(\mathrm{pc}) = w \;\Rightarrow \\
& \left(\mathop{\scalebox{2}{$\ast$}}_{r \in \mathtt{RegName}} r \mapsto_r \mathtt{regs}(r) \,\ast\, \mathcal{V}(\mathtt{regs}(r)) \right) \mathrel{-\!\!\ast} \\
& \mathtt{WP}\ \mathtt{Running}\ \{v,\ v = \mathtt{Halted}\}
\end{aligned}
$$

Binary validity relation

Defined on equal values:

$$
\begin{array}{lll}
\mathcal{V}(z,\ z) & := & \texttt{True} \\
\mathcal{V}((\texttt{O}, b, e, a),\ \_) & := & \texttt{True} \\
\mathcal{V}((\texttt{E}, b, e, a),\ \_) & := & \triangleright \Box\, \mathcal{E}((\texttt{RX}, b, e, a),\ (\texttt{RX}, b, e, a)) \\
\mathcal{V}((\texttt{R}/\texttt{RX}, b, e, a),\ \_) & := & \displaystyle\mathop{\Huge *}_{a \in [b;e)} \exists P, \begin{cases} \boxed{\exists\, w\, w',\ a \mapsto_a w * a \rightarrowtail_a w' * P(w, w')} \ * \\ \triangleright \Box\, \forall\, w\, w',\ P(w, w') \ \twoheadrightarrow \mathcal{V}(w,\ w') \end{cases} \\
\mathcal{V}((\texttt{RW}/\texttt{RWX}, b, e, a),\ \_) & := & \displaystyle\mathop{\Huge *}_{a \in [b;e)} \boxed{\exists\, w\, w',\ a \mapsto_a w \ * \ a \rightarrowtail_a w' \ * \ \mathcal{V}(w,\ w')}
\end{array}
$$

Binary expression relation

$$\begin{aligned}\mathcal{E}\,(w_\ell,\, w_r) \;:=\;& \forall\, \mathtt{regs}_\ell,\, \mathtt{regs}_r,\; \mathtt{regs}_\ell(\mathrm{pc}) = w_\ell \;\wedge\; \mathtt{regs}_r(\mathrm{pc}) = w_r \;\Rightarrow \\ & \left(\mathop{\Huge\ast}_{r \in \mathtt{RegName}} r \mapsto_r \mathtt{regs}_\ell(r) \ast r \rightarrowtail_r \mathtt{regs}_r(r) \ast \mathcal{V}\,(\mathtt{regs}_\ell(r),\, \mathtt{regs}_r(r)) \right) \mathrel{-\!\!\ast} \\ & \mathtt{WP}\;(\mathtt{Running}, \mathtt{Running})\;\{(v_\ell, v_r),\; v_\ell = \mathtt{Halted} \Rightarrow v_r = \mathtt{Halted}\}\end{aligned}$$

Binary expression relation

$$\mathcal{E}\,(w_\ell,\, w_r) \;:=\; \forall\, \mathtt{regs}_\ell,\, \mathtt{regs}_r,\; \mathtt{regs}_\ell(\mathrm{pc}) = w_\ell \;\wedge\; \mathtt{regs}_r(\mathrm{pc}) = w_r \;\Rightarrow$$
$$\left(\mathop{\Huge\ast}_{r \in \mathtt{RegName}} r \mapsto_r \mathtt{regs}_\ell(r) * r \rightarrowtail_r \mathtt{regs}_r(r) * \mathcal{V}\,(\mathtt{regs}_\ell(r),\, \mathtt{regs}_r(r)) \right) \mathrel{-\!\!*}$$
$$\mathtt{WP}\;(\mathtt{Running}, \mathtt{Running})\;\{(v_\ell, v_r),\; v_\ell = \mathtt{Halted} \Rightarrow v_r = \mathtt{Halted}\}$$

$\Rightarrow$ similar implication to the one in contextual refinement

Fundamental theorem on logical relations

If a capability is safe to share, it is safe to execute

Fundamental theorem on logical relations

If a capability is safe to share, it is safe to execute

FTLR

$$\begin{array}{l} \texttt{spec\_ctx} \Rightarrow \\ \mathcal{V}\left((p,b,e,a),\ (p,b,e,a)\right) \Rightarrow \\ \mathcal{E}\left((p,b,e,a),\ (p,b,e,a)\right) \end{array}$$

Exports relation

Goal: link validity (words) to CR (components)

Exports relation

Goal: link validity (words) to CR (components)

Exports relation

$$\mathcal{V}_{\text{exp}}(x, y) := \mathop{\Huge\ast}_{s \mapsto w_r \in y.\texttt{exports}} \exists\, w_\ell,\ s \mapsto w_\ell \in x.\texttt{exports} * \mathcal{V}(w_\ell, w_r)$$

Exports relation

Goal: link validity (words) to CR (components)

Exports relation

$$\mathcal{V}_{\text{exp}}(x, y) := \mathop{\Huge\ast}_{s \mapsto w_r \in y.\texttt{exports}} \exists\, w_\ell,\ s \mapsto w_\ell \in x.\texttt{exports} * \mathcal{V}(w_\ell, w_r)$$

Implies $\texttt{dom}\ y.\texttt{exports} \subseteq \texttt{dom}\ x.\texttt{exports}$

Compatibility with link

Let x, y, z be components such that:

- x and z are disjoint; y and z are disjoint;
- $\mathtt{img}\,(z.\mathtt{segment}) \subseteq \mathbb{Z}$;
- $\mathtt{dom}\ x.\mathtt{exports} \subseteq \mathtt{dom}\ y.\mathtt{exports}$;

Then:

$$\mathtt{spec\_ctx} * \mathcal{V}_{\mathsf{exp}}\,(x,\ y) * \mathsf{mem\_map}_{\ell}(x, z) * \mathsf{mem\_map}_{r}(y, z) \\ \Rrightarrow \quad \mathcal{V}_{\mathsf{exp}}\,(x \ \bowtie\ z,\ y \ \bowtie\ z)$$

Conclusion

Conclusion

Remaining work:

- Show link between $\mathcal{V}_{\text{exp}}$ and CR
- Strenghten theorem on $\mathcal{V}_{\text{exp}}$ of links

Conclusion

Remaining work:

- Show link between $\mathcal{V}_{\text{exp}}$ and CR
- Strenghten theorem on $\mathcal{V}_{\text{exp}}$ of links

Reflexions on CR:

- Too strong relation for many practical cases
- Maybe try to restrict observable behaviors

Thank you for your attention

Questions?