Verification of Uninterpreted and Partially Interpreted Programs

Umang Mathur
Joint work with
Madhusudan Parthasarathy and **Mahesh Viswanathan**

University of Illinois at Urbana Champaign

Table of contents

Introduction

Program Verification

Program verification is undecidable, in general.

However, decidable classes do exist:

- Programs without loops or recursion (straight-line)
- Programs working over finite domains (Boolean programs)
- Models like Petri Nets - not natural for modeling programs

Today : Decidable verification for programs with loops/recursion while working over infinite domains.

Uninterpreted Programs

What are Uninterpreted Programs?

- Programs over an uninterpreted vocabulary
  - Constant, function and relation symbols are *completely uninterpreted*.
- Work over arbitrary data models
  - Data models provide interpretations to symbols in the program.
- Satisfy ϕ if ϕ holds in *all* data models

Uninterpreted Programs: Syntax

Fix a finite set V of program variables.
Fix a first order vocabulary $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{R})$.

Program Syntax

$$
\begin{aligned}
\langle \text{stmt} \rangle ::= \; & \textbf{skip} \mid x := c \mid x := y \mid x := f(\mathbf{z}) \\
& \mid \textbf{if} \, (\langle \text{cond} \rangle) \, \textbf{then} \, \langle \text{stmt} \rangle \, \textbf{else} \, \langle \text{stmt} \rangle \mid \textbf{while} \, (\langle \text{cond} \rangle) \, \langle \text{stmt} \rangle \\
& \mid \textbf{assume} \, (\langle \text{cond} \rangle) \mid \langle \text{stmt} \rangle \, ; \, \langle \text{stmt} \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \text{cond} \rangle ::= \; & \texttt{true} \mid x = y \mid x = c \mid c = d \mid R(\mathbf{z}) \\
& \mid \langle \text{cond} \rangle \vee \langle \text{cond} \rangle \mid \neg \langle \text{cond} \rangle
\end{aligned}
$$

where, $x, y, \mathbf{z} \in V$, $c \in \mathcal{C}$, $f \in \mathcal{F}$ and $R \in \mathcal{R}$.

Example

```
assume (T ≠ F);
b := F;
while (x ≠ y) {
    d := key(x);
    if (d = k) then {
        b := T;
        r := x;
    }
    x := n(x);
}
```

- Searches for an element with key `k` in a list starting at `x` and ending at `y`.
- `T` and `F` are uninterpreted constants
- `key` and `n` are uninterpreted functions

Example

```
assume (T ≠ F);
b := F;
while (x ≠ y) {
   d := key(x);
   if (d = k) then {
      b := T;
      r := x;
   }
   x := n(x);
}
```

- Searches for an element with key `k` in a list starting at `x` and ending at `y`.
- `T` and `F` are uninterpreted constants
- `key` and `n` are uninterpreted functions

Example

```
assume (T ≠ F);
b := F;
while (x ≠ y) {
    d := key(x);
    if (d = k) then {
        b := T;
        r := x;
    }
    x := n(x);
}
```

- Searches for an element with key k in a list starting at x and ending at y.
- T and F are uninterpreted constants
- key and n are uninterpreted functions

Uninterpreted Programs: Executions

Executions are finite sequences over the following alphabet

$$\Pi = \left\{ \begin{array}{l} \text{“}x := y\text{”}, \text{“}x := f(\mathbf{z})\text{”}, \\ \text{“}\mathbf{assume}(x = y)\text{”}, \text{“}\mathbf{assume}(x \neq y)\text{”}, \\ \text{“}\mathbf{assume}(R(\mathbf{z}))\text{”}, \text{“}\mathbf{assume}(\neg R(\mathbf{z}))\text{”} \end{array} \middle| \begin{array}{l} x, y, \mathbf{z} \in V, \\ f \in \mathcal{F}, R \in \mathcal{R} \end{array} \right\}$$

Uninterpreted Programs: Executions

Executions are finite sequences over the following alphabet

$$\Pi = \left\{ \begin{array}{l} \text{“}x := y\text{”}, \text{“}x := f(\mathbf{z})\text{”}, \\ \text{“}\mathbf{assume}(x = y)\text{”}, \text{“}\mathbf{assume}(x \neq y)\text{”}, \\ \text{“}\mathbf{assume}(R(\mathbf{z}))\text{”}, \text{“}\mathbf{assume}(\neg R(\mathbf{z}))\text{”} \end{array} \;\middle|\; \begin{array}{l} x, y, \mathbf{z} \in V, \\ f \in \mathcal{F}, R \in \mathcal{R} \end{array} \right\}$$

Set of executions is a regular language defined inductively:

$$\begin{array}{lcl}
\mathsf{Exec}(\mathbf{skip}) & = & \{\epsilon\} \\
\mathsf{Exec}(x := y) & = & \{\text{“}x := y\text{”}\} \\
\mathsf{Exec}(x := f(\mathbf{z})) & = & \{\text{“}x := f(\mathbf{z})\text{”}\} \\
\mathsf{Exec}(\mathbf{assume}(c)) & = & \{\text{“}\mathbf{assume}(c)\text{”}\} \\
\mathsf{Exec}(\mathbf{if}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) & = & \begin{array}{l} \{\text{“}\mathbf{assume}(c)\text{”}\} \cdot \mathsf{Exec}(s_1) \\ \cup \{\text{“}\mathbf{assume}(\neg c)\text{”}\} \cdot \mathsf{Exec}(s_2) \end{array} \\
\mathsf{Exec}(s_1; s_2) & = & \mathsf{Exec}(s_1) \cdot \mathsf{Exec}(s_2) \\
\mathsf{Exec}(\mathbf{while}\ c\ \{s\}) & = & \Big(\{\text{“}\mathbf{assume}(c)\text{”}\} \cdot \mathsf{Exec}(s)\Big)^* \cdot \{\text{“}\mathbf{assume}(\neg c)\text{”}\}
\end{array}$$

Semantics given by a first order structure $\mathtt{M} = (\mathcal{U}_{\mathtt{M}}, [\![]\!]_{\mathtt{M}})$ on Σ.

Definition (Values of Variables)

$$
\begin{array}{rcll}
\mathsf{val}_{\mathtt{M}}(\epsilon, x) & = & [\![\widehat{x}]\!]_{\mathtt{M}} & \text{for every } x \in V \\
\mathsf{val}_{\mathtt{M}}(\rho\cdot\text{``}x := y\text{''}, z) & = & \mathsf{val}_{\mathtt{M}}(\rho, y) & \text{if } z \text{ is } x \\
\mathsf{val}_{\mathtt{M}}(\rho\cdot\text{``}x := f(z_1, \ldots)\text{''}, y) & = & [\![f]\!]_{\mathtt{M}}(\mathsf{val}_{\mathtt{M}}(\rho, z_1), \ldots) & \text{if } y \text{ is } x \\
\mathsf{val}_{\mathtt{M}}(\rho\cdot a, x) & = & \mathsf{val}_{\mathtt{M}}(\rho, x) & \text{otherwise}
\end{array}
$$

Uninterpreted Programs: Semantics

Semantics given by a first order structure $\mathtt{M} = (\mathcal{U}_{\mathtt{M}}, [\![]\!]_{\mathtt{M}})$ on Σ.

Definition (Feasibility of Execution)

An execution ρ is feasible in $\mathtt{M}$ if for every prefix $\sigma' = \sigma \cdot$ "**assume**(c)" of ρ, we have

1. $\mathsf{val}_{\mathtt{M}}(\sigma, x) = \mathsf{val}_{\mathtt{M}}(\sigma, y)$ if c is $(x = y)$,
2. $\mathsf{val}_{\mathtt{M}}(\sigma, x) \neq \mathsf{val}_{\mathtt{M}}(\sigma, y)$ if c is $(x \neq y)$,
3. $(\mathsf{val}_{\mathtt{M}}(\sigma, z_1), \ldots, \mathsf{val}_{\mathtt{M}}(\sigma, z_r)) \in [\![R]\!]_{\mathtt{M}}$ if c is $R(z_1, \ldots, z_r)$, and
4. $(\mathsf{val}_{\mathtt{M}}(\sigma, z_1), \ldots, \mathsf{val}_{\mathtt{M}}(\sigma, z_r)) \notin [\![R]\!]_{\mathtt{M}}$ if c is $\neg R(z_1, \ldots, z_r)$.

Uninterpreted Programs: Verification

Definition (Verification of Uninterpreted Programs)

Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

Uninterpreted Programs: Verification

Definition (Verification of Uninterpreted Programs)

Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$

Definition (Verification of Uninterpreted Programs)

Let $P \in \langle \mathsf{stmt} \rangle$ be an uninterpreted program and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ iff for every execution $\rho \in \mathsf{Exec}(P)$

Uninterpreted Programs: Verification

Definition (Verification of Uninterpreted Programs)

Let $P \in \langle\text{stmt}\rangle$ be an uninterpreted program and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ iff for every execution $\rho \in \text{Exec}(P)$ and for every FO structure $\mathtt{M}$ such that ρ is feasible in $\mathtt{M}$,

Uninterpreted Programs: Verification

Definition (Verification of Uninterpreted Programs)

Let $P \in \langle \mathsf{stmt} \rangle$ be an uninterpreted program and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ iff for every execution $\rho \in \mathsf{Exec}(P)$ and for every FO structure $\mathtt{M}$ such that ρ is feasible in $\mathtt{M}$, $\mathtt{M}$ satisfies $\varphi[\mathsf{val}_{\mathtt{M}}(\rho, V)/V]$.

Uninterpreted Programs: Verification

Definition (Verification of Uninterpreted Programs)

Let $P \in \langle \mathsf{stmt} \rangle$ be an uninterpreted program and let φ be an assertion in the following grammar.

$$\varphi ::= \mathtt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ iff for every execution $\rho \in \mathsf{Exec}(P)$ and for every FO structure $\mathtt{M}$ such that ρ is feasible in $\mathtt{M}$, $\mathtt{M}$ satisfies $\varphi[\mathsf{val}_{\mathtt{M}}(\rho, V)/V]$.

Theorem [1, 3]

Verification of uninterpreted programs is undecidable.

Coherence

How do we verify a single execution?

Execution ρ

$$
\begin{gathered}
\textbf{assume}(\mathtt{T} \neq \mathtt{F}) \\
\mathtt{b} := \mathtt{F} \\
\textbf{assume}(\mathtt{x} \neq \mathtt{y}) \\
\mathtt{d} := \mathtt{key(x)} \\
\textbf{assume}(\mathtt{d} = \mathtt{k}) \\
\mathtt{b} := \mathtt{T} \\
\mathtt{r} := \mathtt{x} \\
\mathtt{x} := \mathtt{n(x)} \\
\textbf{assume}(\mathtt{x} = \mathtt{y})
\end{gathered}
$$

$\varphi \equiv \mathtt{b{=}T} \Rightarrow \mathtt{key(r){=}k}$

How do we verify a single execution?

Execution ρ

$$\textbf{assume}(\mathtt{T} \neq \mathtt{F})$$
$$\mathtt{b} := \mathtt{F}$$
$$\textbf{assume}(\mathtt{x} \neq \mathtt{y})$$
$$\mathtt{d} := \mathtt{key(x)}$$
$$\textbf{assume}(\mathtt{d} = \mathtt{k})$$
$$\mathtt{b} := \mathtt{T}$$
$$\mathtt{r} := \mathtt{x}$$
$$\mathtt{x} := \mathtt{n(x)}$$
$$\textbf{assume}(\mathtt{x} = \mathtt{y})$$

$$\varphi \equiv \mathtt{b{=}T} \Rightarrow \mathtt{key(r){=}k}$$

$VC(\rho, \varphi)$

$$\begin{array}{lc}
 & \mathtt{T} \neq \mathtt{F} \\
\wedge & \mathtt{b}_1 = \mathtt{F} \\
\wedge & \mathtt{x}_0 \neq \mathtt{y}_0 \\
\wedge & \mathtt{d}_1 = \mathtt{key}(\mathtt{x}_0) \\
\wedge & \mathtt{d}_1 = \mathtt{k}_0 \\
\wedge & \mathtt{b}_2 = \mathtt{T} \\
\wedge & \mathtt{r}_1 = \mathtt{x}_0 \\
\wedge & \mathtt{x}_1 = \mathtt{n}(\mathtt{x}_0) \\
\wedge & \mathtt{x}_1 = \mathtt{y}_0 \\
 & \\
\Rightarrow & (\mathtt{b}_2 = \mathtt{T} \Rightarrow \mathtt{key}(\mathtt{r}_1) = \mathtt{k}_0)
\end{array}$$

How do we verify a single execution?

Execution ρ

$$
\begin{array}{c}
\textbf{assume}(\mathtt{T} \neq \mathtt{F}) \\
\mathtt{b} := \mathtt{F} \\
\textbf{assume}(\mathtt{x} \neq \mathtt{y}) \\
\mathtt{d} := \mathtt{key(x)} \\
\textbf{assume}(\mathtt{d} = \mathtt{k}) \\
\mathtt{b} := \mathtt{T} \\
\mathtt{r} := \mathtt{x} \\
\mathtt{x} := \mathtt{n(x)} \\
\textbf{assume}(\mathtt{x} = \mathtt{y})
\end{array}
$$

$$\varphi \equiv \mathtt{b{=}T} \Rightarrow \mathtt{key(r){=}k}$$

$VC(\rho, \varphi)$

$$
\begin{array}{lc}
 & \mathtt{T} \neq \mathtt{F} \\
\wedge & \mathtt{b}_1 = \mathtt{F} \\
\wedge & \mathtt{x}_0 \neq \mathtt{y}_0 \\
\wedge & \mathtt{d}_1 = \mathtt{key}(\mathtt{x}_0) \\
\wedge & \mathtt{d}_1 = \mathtt{k}_0 \\
\wedge & \mathtt{b}_2 = \mathtt{T} \\
\wedge & \mathtt{r}_1 = \mathtt{x}_0 \\
\wedge & \mathtt{x}_1 = \mathtt{n}(\mathtt{x}_0) \\
\wedge & \mathtt{x}_1 = \mathtt{y}_0 \\
 & \\
\Rightarrow & (\mathtt{b}_2 = \mathtt{T} \Rightarrow \mathtt{key}(\mathtt{r}_1) = \mathtt{k}_0)
\end{array}
$$

φ holds in every $\mathtt{M}$ in which ρ is feasible

iff

$VC(\rho, \varphi)$ is valid in T_{EUF}

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.
  - Congruence closure algorithm

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.
  - Congruence closure algorithm
  - Polynomial time when φ is a single atom.

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.
  - Congruence closure algorithm
  - Polynomial time when φ is a single atom.
- But programs have infinitely many executions.

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.
  - Congruence closure algorithm
  - Polynomial time when φ is a single atom.
- But programs have infinitely many executions.
- How do we recover decidability?

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.
  - Congruence closure algorithm
  - Polynomial time when φ is a single atom.
- But programs have infinitely many executions.
- How do we recover decidability?
- Coherence to the rescue!

How do we verify a single execution?

- Verification of *a single execution* can be reduced to checking validity of a quantifier-free formula in T_{EUF}.
  - Congruence closure algorithm
  - Polynomial time when φ is a single atom.
- But programs have infinitely many executions.
- How do we recover decidability?
- Coherence to the rescue!
  - Allows congruence closure to be performed in a *streaming* fashion.

Congruence Closure

Congruence on Ground Terms

Let $\Sigma = (\mathcal{C}, \mathcal{F})$ be a FO-vocabulary. Let $t_1, t_1', t_2, \ldots, t_k, t_k'$ be ground terms on Σ and let $f \in \mathcal{F}$ be a k-ary function. Then,

$$\frac{t_1 = t_1' \qquad t_2 = t_2' \qquad \ldots \qquad t_k = t_k'}{f(t_1, t_2, \ldots, t_k) = f(t_1', t_2', \ldots, t_k')}$$

Congruence Closure

Congruence on Ground Terms

Let $\Sigma = (\mathcal{C}, \mathcal{F})$ be a FO-vocabulary. Let $t_1, t'_1, t_2, \ldots, t_k, t'_k$ be ground terms on Σ and let $f \in \mathcal{F}$ be a k-ary function. Then,

$$\frac{t_1 = t'_1 \qquad t_2 = t'_2 \qquad \ldots \qquad t_k = t'_k}{f(t_1, t_2, \ldots, t_k) = f(t'_1, t'_2, \ldots, t'_k)}$$

Interpretation

In every FO structure $\mathtt{M}$,

$$\begin{array}{rc} \text{if} & [\![t_1]\!]_{\mathtt{M}} = [\![t'_1]\!]_{\mathtt{M}},\ [\![t_2]\!]_{\mathtt{M}} = [\![t'_2]\!]_{\mathtt{M}},\ \ldots,\ \text{and } [\![t_k]\!]_{\mathtt{M}} = [\![t'_k]\!]_{\mathtt{M}} \\ \text{then} & [\![f(t_1, t_2, \ldots, t_k)]\!]_{\mathtt{M}} = [\![f(t'_1, t'_2, \ldots, t'_k)]\!]_{\mathtt{M}} \end{array}$$

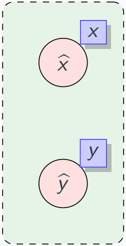
Congruence Closure on Executions

$$\textbf{assume}(x = y) \longrightarrow x_1 := f(x) \longrightarrow y_1 := f(y)$$

Congruence Closure on Executions

$\textbf{assume}(x = y) \longrightarrow x_1 := f(x) \longrightarrow y_1 := f(y)$

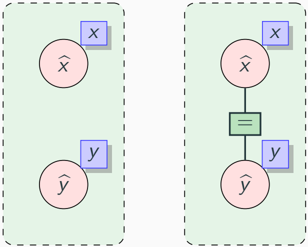
Initially



Congruence Closure on Executions

$\textbf{assume}(x = y) \longrightarrow x_1 := f(x) \longrightarrow y_1 := f(y)$

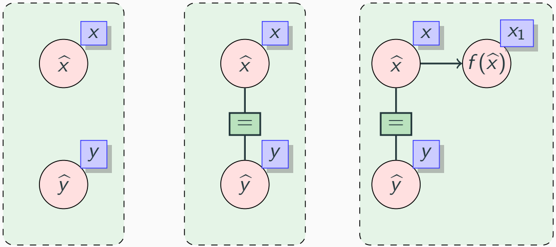
Initially



Congruence Closure on Executions

$\textbf{assume}(x = y) \longrightarrow x_1 := f(x) \longrightarrow y_1 := f(y)$

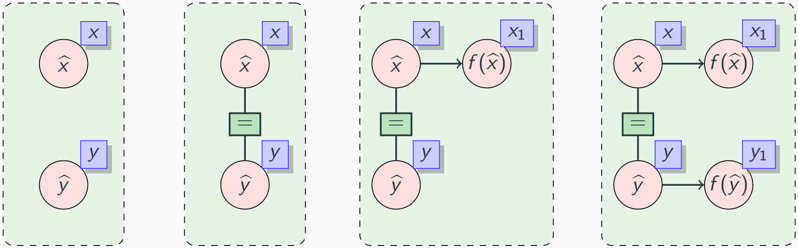
Initially



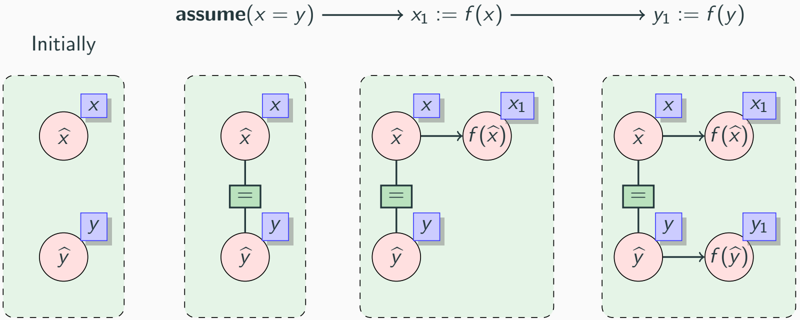
Congruence Closure on Executions

$\textbf{assume}(x = y) \longrightarrow x_1 := f(x) \longrightarrow y_1 := f(y)$

Initially



Congruence Closure on Executions

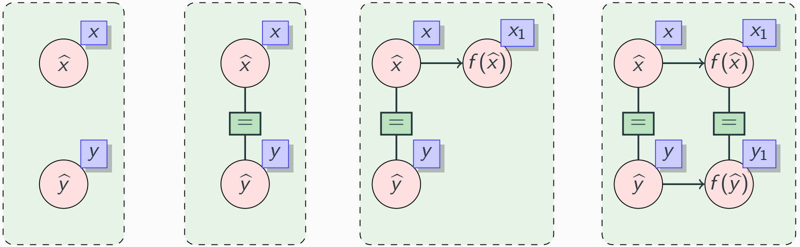


$\varphi : x_1 = y_1$

Congruence Closure on Executions

$\textbf{assume}(x = y) \longrightarrow x_1 := f(x) \longrightarrow y_1 := f(y)$

Initially



$\varphi : x_1 = y_1$

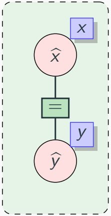
φ holds
after the execution

Congruence Closure on Executions

$$\textbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \xrightarrow{\qquad\qquad} y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$

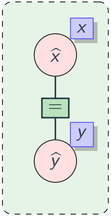
Congruence Closure on Executions

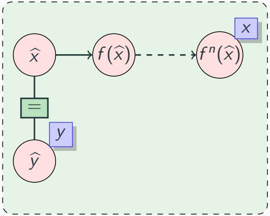
$$\textbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$



Congruence Closure on Executions

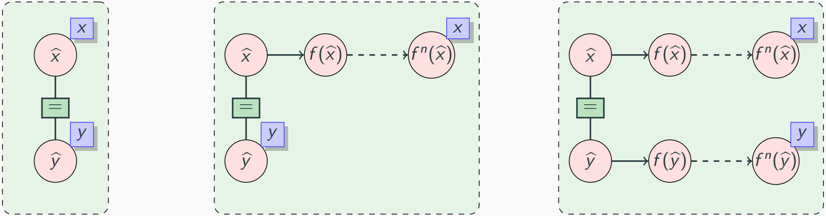
$$\mathbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$





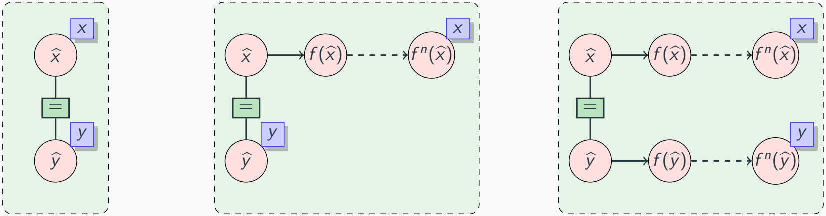
Congruence Closure on Executions

$$\mathbf{assume}(x = y) \longrightarrow x := f(x) \xrightarrow{n \text{ times}} x := f(x) \longrightarrow y := f(y) \xrightarrow{n \text{ times}} y := f(y)$$



Congruence Closure on Executions

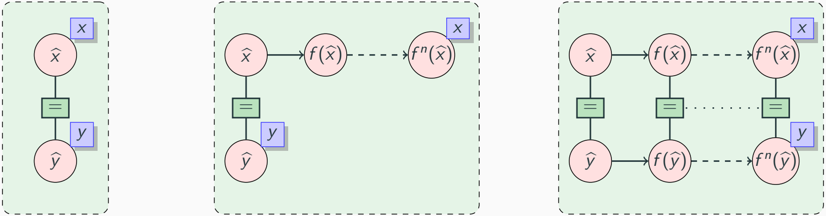
$$\textbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$



$\varphi : \ x = y$

Congruence Closure on Executions

$$\textbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$

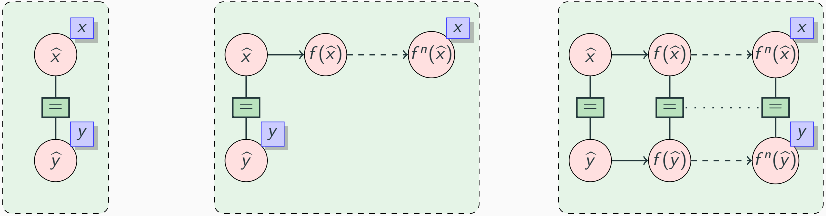


$\varphi : x = y$

φ holds
after the execution

Congruence Closure on Executions

$$\textbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$



$\varphi : x = y$

φ holds
after the execution

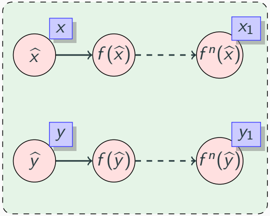
Unbounded memory required to infer equality relationships in a streaming setting.

Congruence Closure on Executions

$$x_1 := f(x) \rightarrow y_1 := f(y) \overset{n \text{ times}}{\dashrightarrow} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$

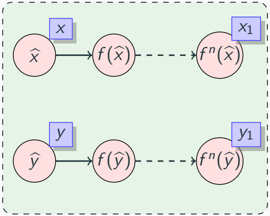
Congruence Closure on Executions

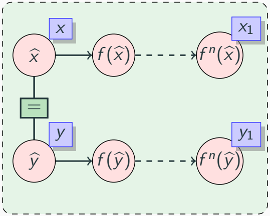
$$x_1 := f(x) \rightarrow y_1 := f(y) \xrightarrow{n \text{ times}} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$



Congruence Closure on Executions

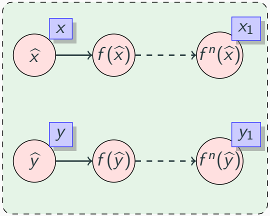
$$x_1 := f(x) \to y_1 := f(y) \overset{n \text{ times}}{\dashrightarrow} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$

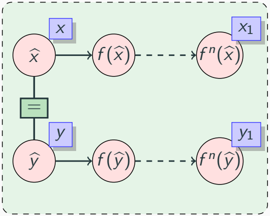




Congruence Closure on Executions

$$x_1 := f(x) \rightarrow y_1 := f(y) \overset{n \text{ times}}{\dashrightarrow} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$

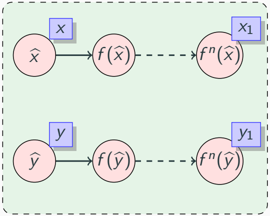


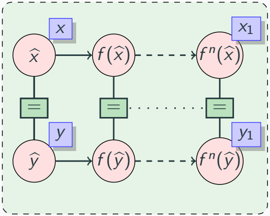


$\varphi :\ x_1 = y_1$

Congruence Closure on Executions

$$x_1 := f(x) \rightarrow y_1 := f(y) \xrightarrow{n \text{ times}} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$



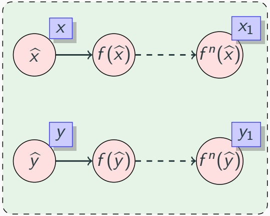


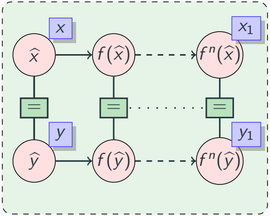
$\varphi : x_1 = y_1$

φ holds
after the execution

Congruence Closure on Executions

$$x_1 := f(x) \rightarrow y_1 := f(y) \xrightarrow{n \text{ times}} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$





$\varphi :\ x_1 = y_1$

φ holds
after the execution

Again, unbounded memory required to infer equality relationships in a streaming setting.

Algebraic View of Executions

Terms Computed

$$\begin{aligned}
\mathrm{Term}(\epsilon, x) &= \widehat{x} && \text{for every } x \in V \\
\mathrm{Term}(\rho \cdot \text{“}x := y\text{”}, z) &= \mathrm{Term}(\rho, y) && \text{if } z \text{ is } x \\
\mathrm{Term}(\rho \cdot \text{“}x := f(z_1, \ldots)\text{”}, y) &= f(\mathrm{Term}(\rho, z_1), \ldots) && \text{if } y \text{ is } x \\
\mathrm{Term}(\rho \cdot a, x) &= \mathrm{Term}(\rho, x) && \text{otherwise}
\end{aligned}$$

Algebraic View of Executions

Terms Computed

$$
\begin{array}{rcll}
\text{Term}(\epsilon, x) & = & \widehat{x} & \text{for every } x \in V \\
\text{Term}(\rho \cdot \text{“}x := y\text{”}, z) & = & \text{Term}(\rho, y) & \text{if } z \text{ is } x \\
\text{Term}(\rho \cdot \text{“}x := f(z_1, \ldots)\text{”}, y) & = & f(\text{Term}(\rho, z_1), \ldots) & \text{if } y \text{ is } x \\
\text{Term}(\rho \cdot a, x) & = & \text{Term}(\rho, x) & \text{otherwise}
\end{array}
$$

Equalities

$$
\begin{aligned}
\alpha(\varepsilon) &= \varnothing \\
\alpha(\rho \cdot \text{“}\textbf{assume}(x = y)\text{”}) &= \alpha(\rho) \cup \{(\text{Term}(\rho, x), \text{Term}(\rho, y))\} \\
\alpha(\rho \cdot a) &= \alpha(\rho) \quad \text{otherwise}
\end{aligned}
$$

Algebraic View of Executions

Terms Computed

$$
\begin{aligned}
\mathrm{Term}(\epsilon, x) &= \widehat{x} && \text{for every } x \in V \\
\mathrm{Term}(\rho \cdot \text{“}x := y\text{”}, z) &= \mathrm{Term}(\rho, y) && \text{if } z \text{ is } x \\
\mathrm{Term}(\rho \cdot \text{“}x := f(z_1, \ldots)\text{”}, y) &= f(\mathrm{Term}(\rho, z_1), \ldots) && \text{if } y \text{ is } x \\
\mathrm{Term}(\rho \cdot a, x) &= \mathrm{Term}(\rho, x) && \text{otherwise}
\end{aligned}
$$

Equalities

$$
\begin{aligned}
\alpha(\varepsilon) &= \varnothing \\
\alpha(\rho \cdot \text{“}\textbf{assume}(x = y)\text{”}) &= \alpha(\rho) \cup \{(\mathrm{Term}(\rho, x), \mathrm{Term}(\rho, y))\} \\
\alpha(\rho \cdot a) &= \alpha(\rho) \quad \text{otherwise}
\end{aligned}
$$

Disequalities

$$
\begin{aligned}
\beta(\varepsilon) &= \varnothing \\
\beta(\rho \cdot \text{“}\textbf{assume}(x \neq y)\text{”}) &= \beta(\rho) \cup \{(\mathrm{Term}(\rho, x), \mathrm{Term}(\rho, y))\} \\
\beta(\rho \cdot a) &= \beta(\rho) \quad \text{otherwise}
\end{aligned}
$$

Coherence

An execution is coherent if it is memoizing and has early assumes.

Coherence = Memoizing + Early Assumes

Coherence

An execution is coherent if it is memoizing and has early assumes.

Coherence = Memoizing + Early Assumes

Remember terms
until re-computation

Equality assumes
occur before
superterms are forgotten

Coherence: Memoizing

Definition (Memoizing Execution)
An execution ρ is memoizing if for every prefix of ρ of the form

$$\sigma' = \sigma \cdot \text{“}x := f(y_1, \ldots, y_r)\text{”}$$

we have the following.

If there is a term $t \in \mathrm{ComputedTerms}(\sigma)$ such that $t \cong_{\alpha(\sigma)} \mathrm{Term}(\sigma', x)$, then there is a variable $z \in V$ such that $\mathrm{Term}(\sigma, z) \cong_{\alpha(\sigma)} \mathrm{Term}(\sigma', x)$.

Here,

- $\mathrm{ComputedTerms}(\sigma) = \{\mathrm{Term}(\pi, v) \mid v \in V, \pi \text{ is a prefix of } \sigma\}$,
- $\cong_{\alpha(\rho)}$ is the smallest congruence induced by $\alpha(\rho)$.

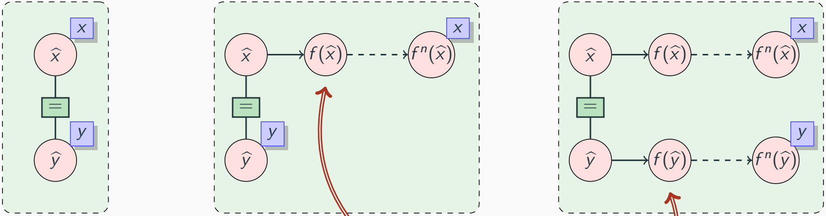
Coherence: Memoizing

```
assume (T ≠ F);
b := F;
while (x ≠ y) {
   d := key(x);
   if (d = k) then {
      b := T;
      r := x;
   }
   x := n(x);
}
```

- All executions of this program are *vacuously* memoizing.
- No term is recomputed.

Example exeuction: Non Memoizing

$$\textbf{assume}(x = y) \longrightarrow x := f(x) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} y := f(y)$$



Re-computation of terms deemed equivalent by $\widehat{x} = \widehat{y}$.
The older term $f(\widehat{x})$ has been dropped.

NOT a memoizing execution

Example exeuction: Memoizing

$$\textbf{assume}(x = y) \longrightarrow x := f(x) \longrightarrow y := f(y) \overset{n \text{ times}}{\dashrightarrow} x := f(x) \longrightarrow y := f(y)$$

Re-computation happens in tandem
(at least one older equivalent terms is available
in some variable at the time of re-computation)

✓ memoizing execution

Definition (Early Assumes)

An execution ρ is said to have early assumes if for every prefix of ρ of the form

$$\sigma' = \sigma \cdot \text{“}\textbf{assume}(x = y)\text{”}$$

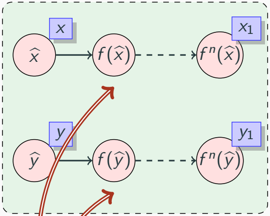
we have the following.

If there is a term $s \in \mathrm{ComputedTerms}(\sigma)$ such that s is a $\alpha(\sigma)$-*superterm* of either $\mathrm{Term}(\sigma, x)$ or $\mathrm{Term}(\sigma, y)$, then there is a variable $z \in V$ such that $\mathrm{Term}(\sigma, z) \cong_{\alpha(\sigma)} s$.

Here, t_1 is a $\alpha(\sigma)$-superterm of t_2 if there are terms t_1' and t_2' such that t_1' is a superterm of t_2', $t_1 \cong_{\alpha(\sigma)} t_1'$ and $t_2 \cong_{\alpha(\sigma)} t_2'$.

Example exeuction: Violation of Early Assumes

$$x_1 := f(x) \to y_1 := f(y) \overset{n \text{ times}}{\dashrightarrow} x_1 := f(x) \longrightarrow y_1 := f(y_1) \longrightarrow \textbf{assume}(x = y)$$



Superterms of $\widehat{x}$ and $\widehat{y}$
dropped before equality assume.

Does NOT satisfy early assumes

Example exeuction: Early Assumes

$$\textbf{assume}(x = y) \longrightarrow x := f(x) \longrightarrow y := f(y) \xrightarrow{n \text{ times}} x := f(x) \longrightarrow y := f(y)$$

✓ Early Assume

Coherence

```
assume (T ≠ F);
b := F;
while (x ≠ y) {
   d := key(x);
   if (d = k) then {
      b := T;
      r := x;
   }
   x := n(x);
}
```

- In every execution, equality assume **assume**($x = y$) occurs on terms without any superterms.
- All executions are coherent!

Coherent Programs and their Verification

An uninterpreted program $P \in \langle \mathsf{stmt} \rangle$ is coherent if all executions of P are coherent.

Coherent Programs and their Verification

An uninterpreted program $P \in \langle \mathsf{stmt} \rangle$ is coherent if all executions of P are coherent.

Decidability of Verification of Coherent Programs [1]

Verification of uninterpreted coherent programs is PSPACE-complete.

Proof.

- Regular language $L^{\varphi}_{\text{coherent}}$ such that for any coherent execution ρ,

$$\rho \in L^{\varphi}_{\text{coherent}} \text{ iff } \rho \models \varphi$$

- The question $\mathsf{Exec}(P) \subseteq L^{\varphi}_{\text{coherent}}$ is decidable.

Regularity of Feasible Coherent Executions

- $P \models \varphi$ iff $P^{\neg\varphi} \models \texttt{false}$, where $P^{\neg\varphi} = P; \textbf{assume}(\neg\varphi)$
- Regular language $L_{\text{coh-feas}}$ such that for any coherent execution ρ,

$$\rho \in L_{\text{coh-feas}} \text{ iff } \rho \text{ is feasible in some FO-structure } \mathtt{M}$$

- $P \models \varphi$ iff $\mathsf{Exec}(P^{\neg\varphi}) \cap L_{\text{coh-feas}} = \varnothing$

Streaming Congruence Closure

- $\mathcal{A}_{\text{coh-feas}} = (Q \uplus \{q_{\text{reject}}\}, q_0, \delta)$ with $L(\mathcal{A}_{\text{coh-feas}}) = L_{\text{coh-feas}}$.
- All states in Q are accepting.
- q_{reject} is absorbing reject state, represents an infeasible execution.
- States in Q are triplets:

$$(\sim,\ d,\ F)$$

Equivalence on variables

$[x_1, x_3]$
$[x_2, x_4, x_5]$
$\vdots$

Disequalities b/w eq. classes

$[x_1, x_3] \neq [x_6]$
$\vdots$

Partial func. relationships b/w eq. classes

$f([x_1, x_3]) = [x_2, x_4, x_5]$
$\vdots$

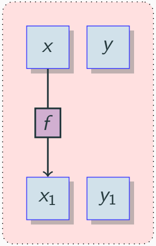
Streaming Congruence Closure

Transitions δ update these relationships in a streaming fashion.

Streaming Congruence Closure

Transitions δ update these relationships in a streaming fashion.

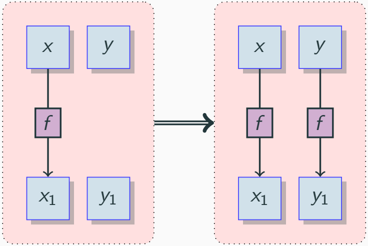
$x_1 = f(x)$



Streaming Congruence Closure

Transitions δ update these relationships in a streaming fashion.

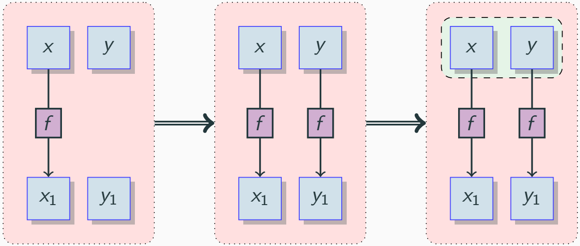
$$x_1 = f(x) \longrightarrow y_1 = f(y)$$



Streaming Congruence Closure

Transitions δ update these relationships in a streaming fashion.

$$x_1 = f(x) \longrightarrow y_1 = f(y) \longrightarrow \textbf{assume}(x = y)$$

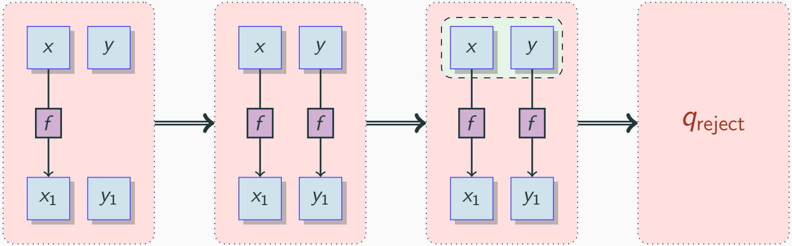


Congruence Closure

Streaming Congruence Closure

Transitions δ update these relationships in a streaming fashion.

$$x_1 = f(x) \longrightarrow y_1 = f(y) \longrightarrow \textbf{assume}(x = y) \longrightarrow \textbf{assume}(x \neq y)$$



Congruence Closure

Streaming Congruence Closure

Correctness of $\mathcal{A}_{\text{coh-feas}}$

Let $\rho \in \Pi^*$ be a coherent execution. Let $q = \delta^*(q_0, \rho)$. Then,

- If ρ is not feasible in any $\mathtt{M}$, then $q = q_{\text{reject}}$
- Otherwise, $q = (\sim, d, P)$ with
  - $\text{Term}(\rho, x) \cong_{\alpha(\rho)} \text{Term}(\rho, y)$ iff $[x]_\sim = [y]_\sim$.
  - $([x]_\sim, [y]_\sim) \in d$ iff there is $(t_x, t_y) \in \beta(\rho)$ such that $t_x \cong_{\alpha(\rho)} \text{Term}(\rho, x)$ and $t_y \cong_{\alpha(\rho)} \text{Term}(\rho, y)$.
  - $f(\text{Term}(\rho, x)) \cong_{\alpha(\rho)} \text{Term}(\rho, y)$ iff $F(f)([x]_\sim) = [y]_\sim$

Checking Coherence

Decidability of Checking Coherence [1]

There is a DFA $\mathcal{A}_{\text{check-coh}}$ such that for an execution $\rho \in \Pi^*$, we have

$$\rho \in L(\mathcal{A}_{\text{check-coh}}) \text{ iff } \rho \text{ is coherent}$$

Checking Coherence

Decidability of Checking Coherence [1]

There is a DFA $\mathcal{A}_{\text{check-coh}}$ such that for an execution $\rho \in \Pi^*$, we have

$$\rho \in L(\mathcal{A}_{\text{check-coh}}) \text{ iff } \rho \text{ is coherent}$$

- $\mathcal{A}_{\text{check-coh}}$ ignores all letters of the form "**assume**$(x \neq y)$".
- States of $\mathcal{A}_{\text{check-coh}}$ maintain $(\sim, F, B)$:
  - $\sim$ and F are as in $\mathcal{A}_{\text{coh-feas}}$
  - B keeps track of the following information: for a given equiv. class c and for a function f, if $f(c)$ has been computed before.

k-Coherence

k-Coherence

```
assume (x ≠ z);
y := n(x);
assume (y ≠ z);

y := n(y);
while (y ≠ z) {
    x := n(x);

    y := n(y);
}

φ ≡ z = n(n(x))
```

k-Coherence

```
assume (x ≠ z);
y := n(x);        ----> n(x̂)
assume (y ≠ z);

y := n(y);        ----> n(n(x̂))
while (y ≠ z) {
    x := n(x);

    y := n(y);
}

φ ≡ z = n(n(x))
```

NOT coherent

- Re-computation without storing prior equivalent terms.
- Insufficient number of program variables to store intermediate terms.

k-Coherence

```
assume (x ≠ z);
y := n(x);
assume (y ≠ z);
g := y;
y := n(y);
while (y ≠ z) {
    x := n(x);
    g := y;
    y := n(y);
}
```

$\varphi \equiv$ z = n(n(x))

1-coherent

- Can be *made coherent*.
- By adding additional ghost variables and assignments to them.
- Write-only and do not change semantics.

k-Coherence

Definition (k-Coherent Executions and Programs)

Let $k \in \mathbb{N}$. Let V be a set of variables and let $G = \{g_1, \ldots, g_k\}$ be additional ghost variables ($V \cap G = \varnothing$).
Let $\Pi_G = \Pi \cup \{\text{“}g := x\text{”} \mid g \in G, x \in V\}$.
An execution over V is k-coherent if there is an execution ρ' over Π_G such that ρ' is coherent and $\rho'\downharpoonright_{\Pi} = \rho$.
A programs is k-coherent if all its executions are.

k-Coherence

Definition (k-Coherent Executions and Programs)

Let $k \in \mathbb{N}$. Let V be a set of variables and let $G = \{g_1, \ldots, g_k\}$ be additional ghost variables ($V \cap G = \varnothing$).
Let $\Pi_G = \Pi \cup \{\text{“}g := x\text{”} \mid g \in G, x \in V\}$.
An execution over V is k-coherent if there is an execution ρ' over Π_G such that ρ' is coherent and $\rho'\downharpoonright_{\Pi} = \rho$.
A programs is k-coherent if all its executions are.

Theorem [1]

Checking k-coherence is decidable in PSPACE. Further, verification of k-coherent programs is decidable in PSPACE.

Verification Modulo Theories

Adding Interpretations

```
assume (T ≠ F);
if (a ≤ b) then {
   if (a ≤ c) then
      min := a;
   else min := c;
}
else {
   if (b ≤ c) then
      min := b;
   else min := c;
}
```

$\varphi \equiv \mathtt{min} \leq \mathtt{a} \wedge \mathtt{min} \leq \mathtt{b}$
$\wedge \mathtt{min} \leq \mathtt{c}$

Find the minimum of a, b and c

Adding Interpretations

```
assume (T ≠ F);
if (a ≤ b) then {
   if (a ≤ c) then
       min := a;
   else min := c;
}
else {
   if (b ≤ c) then
       min := b;
   else min := c;
}
```

$\varphi \equiv \mathtt{min} \leq \mathtt{a} \wedge \mathtt{min} \leq \mathtt{b} \wedge \mathtt{min} \leq \mathtt{c}$

Does not hold in all M.

Find the minimum of a, b and c

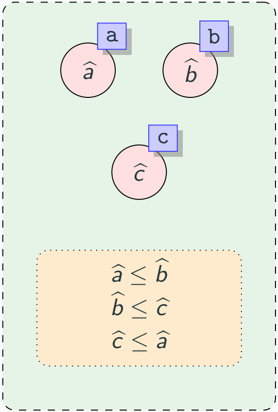
Adding Interpretations

```
assume (T ≠ F);
if (a ≤ b) then {
   if (a ≤ c) then
      min := a;
   else min := c;
}
else {
   if (b ≤ c) then
      min := b;
   else min := c;
}
```

Does not hold in all M.

$\varphi \equiv \texttt{min} \leq \texttt{a} \wedge \texttt{min} \leq \texttt{b} \wedge \texttt{min} \leq \texttt{c}$

Find the minimum of a, b and c



Adding Interpretations

```
assume (T ≠ F);
if (a ≤ b) then {
   if (a ≤ c) then
       min := a;
   else min := c;
}
else {
   if (b ≤ c) then
       min := b;
   else min := c;
}
```

$\varphi \equiv \texttt{min} \leq \texttt{a} \wedge \texttt{min} \leq \texttt{b}$
$\wedge \texttt{min} \leq \texttt{c}$

Find the minimum of a, b and c

This program satisfies φ if $\leq$ is interpreted as a total order:

- $\forall x \cdot x \leq x$
- $\forall x, y, z \cdot x \leq y \wedge y \leq z \implies x \leq z$
- $\forall x, y \cdot x \leq y \wedge y \leq x \implies x = y$

Adding Interpretations

Definition (Verification Modulo Axioms)

Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program over vocabulary Σ. Let A be a set of first order sentences over Σ and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg \varphi$$

Adding Interpretations

Definition (Verification Modulo Axioms)

Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program over vocabulary Σ. Let A be a set of first order sentences over Σ and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ modulo A

Adding Interpretations

Definition (Verification Modulo Axioms)

Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program over vocabulary Σ. Let A be a set of first order sentences over Σ and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ modulo A iff for every execution $\rho \in \text{Exec}(P)$

Adding Interpretations

Definition (Verification Modulo Axioms)

Let $P \in \langle \mathsf{stmt} \rangle$ be an uninterpreted program over vocabulary Σ. Let A be a set of first order sentences over Σ and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ modulo A iff for every execution $\rho \in \mathsf{Exec}(P)$ and for every FO structure $\mathtt{M}$ such that $\mathtt{M} \models A$ and ρ is feasible in $\mathtt{M}$,

Definition (Verification Modulo Axioms)

Let $P \in \langle \mathsf{stmt} \rangle$ be an uninterpreted program over vocabulary Σ. Let A be a set of first order sentences over Σ and let φ be an assertion in the following grammar.

$$\varphi ::= \texttt{true} \mid x = y \mid R(\mathbf{z}) \mid \varphi \vee \varphi \mid \neg\varphi$$

$P \models \varphi$ modulo A iff for every execution $\rho \in \mathsf{Exec}(P)$ and for every FO structure $\mathtt{M}$ such that $\mathtt{M} \models A$ and ρ is feasible in $\mathtt{M}$, $\mathtt{M}$ satisfies $\varphi[\mathsf{val}_{\mathtt{M}}(\rho, V)/V]$.

Coherence Modulo Axioms

Coherence modulo axioms	$=$	Memoizing modulo axioms	$+$	Early Assumes modulo axioms

Example

$$A = \{\forall x, y \cdot f(x, y) = f(y, x)\}$$

Example

$$A = \{\forall x, y \cdot f(x, y) = f(y, x)\}$$

$$x_1 := f(x, y) \longrightarrow y_1 := f(y, x)$$

Example

$$A = \{\forall x, y \cdot f(x, y) = f(y, x)\}$$

$$x_1 := f(x, y) \longrightarrow y_1 := f(y, x)$$

re-computation modulo A

Example

$$A = \{\forall x, y \cdot f(x,y) = f(y,x)\}$$

re-computation modulo A

$$x_1 := f(x,y) \longrightarrow y_1 := f(y,x)$$

$$x_1 := f(x,y) \rightarrow y_1 := f(y,x') \longrightarrow z := g(x_1) \longrightarrow z' := g(y_1) \rightarrow \textbf{assume}(x = x')$$

Example

$A = \{\forall x, y \cdot f(x, y) = f(y, x)\}$

$x_1 := f(x, y) \longrightarrow y_1 := f(y, x)$

re-computation
modulo A

$x_1 := f(x, y) \rightarrow y_1 := f(y, x') \longrightarrow z := g(x_1) \longrightarrow z' := g(y_1) \rightarrow \textbf{assume}(x = x')$

Implied equality
$z = z'$

Memoizing Modulo Axioms

Definition (Memoizing modulo axioms)

Let A be a set of axioms and let $\rho \in \Pi^*$ be an execution. Then, ρ is said to be memoizing modulo A if the following holds.

Let $\sigma' = \sigma \cdot$ "$x := f(\mathbf{z})$" be a prefix of ρ. If there is a term $t' \in \text{ComputedTerms}(\sigma)$ such that $t' \cong_{A \cup \kappa(\sigma)} \text{Term}(\sigma', x)$, then there must exist some variable $y \in V$ such that $\text{Term}(\sigma, y) \cong_{A \cup \kappa(\sigma)} t$.

Memoizing Modulo Axioms

Definition (Memoizing modulo axioms)

Let A be a set of axioms and let $\rho \in \Pi^*$ be an execution. Then, ρ is said to be memoizing modulo A if the following holds.

Let $\sigma' = \sigma \cdot$ "$x := f(\mathbf{z})$" be a prefix of ρ. If there is a term $t' \in \text{ComputedTerms}(\sigma)$ such that $t' \cong_{A \cup \kappa(\sigma)} \text{Term}(\sigma', x)$, then there must exist some variable $y \in V$ such that $\text{Term}(\sigma, y) \cong_{A \cup \kappa(\sigma)} t$.

Here,

$$\begin{aligned}
\kappa(\varepsilon) &= \varnothing \\
\kappa(\rho \cdot \text{“}\textbf{assume}(x = y)\text{”}) &= \kappa(\rho) \cup \{(\text{Term}(\rho, x) = \text{Term}(\rho, y))\} \\
\kappa(\rho \cdot \text{“}\textbf{assume}(x \neq y)\text{”}) &= \kappa(\rho) \cup \{(\text{Term}(\rho, x) \neq \text{Term}(\rho, y))\} \\
\kappa(\rho \cdot \text{“}R(z_1, \ldots)\text{”}) &= \kappa(\rho) \cup \{R(\text{Term}(\rho, z_1), \ldots)\} \\
\kappa(\rho \cdot a) &= \kappa(\rho) \quad \text{otherwise}
\end{aligned}$$

Early Assumes Modulo Axioms

Definition (Early assumes modulo axioms)

Let A be a set of axioms and let $\rho \in \Pi^*$ be an execution. Then, ρ is said to have early assumes modulo A if the following holds.

Let $\sigma' = \sigma \cdot$ "**assume**(c)" be a prefix of ρ, where c is any of $x = y$, $x \neq y$, $R(\mathbf{z})$, or $\neg R(\mathbf{z})$.
Let $t \in \mathrm{ComputedTerms}(\sigma)$ be a term computed in σ such that t has been *dropped*, i.e., for every $x \in V$, we have $\mathrm{Term}(\sigma, x) \ncong_{A \cup \kappa(\sigma)} t$.
For any term $t' \in \mathrm{ComputedTerms}(\sigma)$, if $t \cong_{A \cup \kappa(\sigma')} t'$, then $t \cong_{A \cup \kappa(\sigma)} t'$.

Verification Modulo Axioms - Decidability Landscape [2]

Relational axioms	Decidability
EPR	✗
Reflexivity	✓
Irreflexivity	✓
Symmetry	✓
Transitivity	✓
Partial Order	✓
Total Order	✓

Functional axioms	Decidability
Associativity	✗
Commutativity	✓
Idempotence	✓

Combinations	Decidability
All combinations of decidable axioms	✓

Thank You!

Coherence Modulo Commutativity

Homomorphism h^f_{comm} uses auxiliary variable $v^* \notin V$:

$$h^f_{\text{comm}}(a) = \begin{cases} a \cdot \text{“}v^* := f(y,x)\text{”} \cdot \text{“}\textbf{assume}(z = v^*)\text{”} & \text{if } a = \text{“}z := f(x,y)\text{”} \\ a & \text{otherwise} \end{cases}$$

Coherence Modulo Commutativity

An execution ρ is coherent modulo A iff $h^f_{\text{comm}}(a)$ is coherent modulo $\varnothing$.

Feasibility Modulo Commutativity

An execution ρ is feasible modulo A iff $h^f_{\text{comm}}(a)$ is feasible modulo $\varnothing$.

References I

U. Mathur, P. Madhusudan, and M. Viswanathan.
Decidable verification of uninterpreted programs.
Proc. ACM Program. Lang., 3(POPL), Jan. 2019.

U. Mathur, P. Madhusudan, and M. Viswanathan.
What's decidable about program verification modulo axioms?
In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–177, Cham, 2020. Springer International Publishing.

M. Müller-Olm, O. Rüthing, and H. Seidl.
Checking herbrand equalities and beyond.
In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, pages 79–96, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.