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

1. Introduction

2. Uninterpreted Programs
   - Syntax and Semantics
   - Verification

3. Coherence
   - Verification of Coherent Programs
   - Checking Coherence

4. $k$-Coherence

5. Verification Modulo Theories
Introduction
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 $\phi$ if $\phi$ holds in all data models
Uninterpreted Programs: Syntax

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

Program Syntax

\[
\langle \text{stmt} \rangle ::= \text{skip} \mid x := c \mid x := y \mid x := f(z) \\
\mid \text{if} (\langle \text{cond} \rangle) \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \mid \text{while} (\langle \text{cond} \rangle) \langle \text{stmt} \rangle \\
\mid \text{assume} (\langle \text{cond} \rangle) \mid \langle \text{stmt} \rangle ; \langle \text{stmt} \rangle
\]

\[
\langle \text{cond} \rangle ::= \text{true} \mid x = y \mid x = c \mid c = d \mid R(z) \\
\mid \langle \text{cond} \rangle \lor \langle \text{cond} \rangle \mid \neg \langle \text{cond} \rangle
\]

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

Example

assume \( T \neq F \);
b := F;
while (x \neq 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 \neq F);
b := F;
while (x \neq 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
- \texttt{key} and \texttt{n} are uninterpreted functions
Uninterpreted Programs: Executions

Executions are finite sequences over the following alphabet

\[ \Pi = \left\{ \begin{align*}
&\text{“} x := y \text{”}, \text{“} x := f(z) \text{”}, \\
&\text{“} \text{assume}(x = y) \text{”}, \text{“} \text{assume}(x \neq y) \text{”}, \\
&\text{“} \text{assume}(R(z)) \text{”}, \text{“} \text{assume}(\neg R(z)) \text{”} \\
&\end{align*} \right\} \quad \begin{align*}
x, y, z &\in V, \\
f &\in F, R &\in R
\end{align*} \]
Uninterpreted Programs: Executions

Executions are finite sequences over the following alphabet

$\Pi = \left\{ \text{"x := y", "x := f(z)"}, \right.$

$\left\{ \text{"assume(x = y)"}, \text{"assume(x \neq y)"}, \right.$

$\left\{ \text{"assume(R(z))"}, \text{"assume(\neg R(z))"} \right\}$

$\middle| \begin{align*}
  & x, y, z \in V, \\
  & f \in \mathcal{F}, R \in \mathcal{R}
\end{align*} \right\}$

Set of executions is a regular language defined inductively:

- $\text{Exec(skip)} = \{ \epsilon \}$
- $\text{Exec(x := y)} = \{ "x := y" \}$
- $\text{Exec(x := f(z))} = \{ "x := f(z)" \}$
- $\text{Exec(assume(c))} = \{ "\text{assume(c)}" \}$
- $\text{Exec(if \ c \ then \ s_1 \ else \ s_2)} = \{ "\text{assume(c)}" \} \cdot \text{Exec(s_1)}$
- $\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
Semantics given by a first order structure $M = (U_M, \mathbb{M})$ on $\Sigma$.

**Definition (Values of Variables)**

\[
\begin{align*}
\text{val}_M(\epsilon, x) &= \mathbb{M}[\hat{x}]_M & \text{for every } x \in V \\
\text{val}_M(\rho \cdot "x := y", z) &= \text{val}_M(\rho, y) & \text{if } z \text{ is } x \\
\text{val}_M(\rho \cdot "x := f(z_1, \ldots)", y) &= \mathbb{M}[f]_M(\text{val}_M(\rho, z_1), \ldots) & \text{if } y \text{ is } x \\
\text{val}_M(\rho \cdot a, x) &= \text{val}_M(\rho, x) & \text{otherwise}
\end{align*}
\]
Semantics given by a first order structure $M = (\mathcal{U}_M, []_M)$ on $\Sigma$.

**Definition (Feasibility of Execution)**
An execution $\rho$ is **feasible** in $M$ if for every prefix $\sigma' = \sigma \cdot \text{“}\text{assume}(c)\text{”}$ of $\rho$, we have

1. $\text{val}_M(\sigma, x) = \text{val}_M(\sigma, y)$ if $c$ is $(x = y)$,
2. $\text{val}_M(\sigma, x) \neq \text{val}_M(\sigma, y)$ if $c$ is $(x \neq y)$,
3. $(\text{val}_M(\sigma, z_1), \ldots, \text{val}_M(\sigma, z_r)) \in \llbracket R \rrbracket_M$ if $c$ is $R(z_1, \ldots, z_r)$, and
4. $(\text{val}_M(\sigma, z_1), \ldots, \text{val}_M(\sigma, z_r)) \not\in \llbracket R \rrbracket_M$ if $c$ is $\neg R(z_1, \ldots, z_r)$. 

Definition (Verification of Uninterpreted Programs)

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

\[
\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi
\]
Definition (Verification of Uninterpreted Programs)

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

$$
\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi
$$

$P \models \varphi$
Definition (Verification of Uninterpreted Programs)

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

$$\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \land \varphi \mid \neg \varphi$$

$P \models \varphi$ iff for every execution $\rho \in \text{Exec}(P)$
Definition (Verification of Uninterpreted Programs)
Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program and let $\varphi$ be an assertion in the following grammar.

\[
\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi
\]

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

$M \models_{val_{M}(\rho, V)} \varphi$. 

Theorem [1, 3]
Verification of uninterpreted programs is undecidable.
Definition (Verification of Uninterpreted Programs)

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

$$\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi$$

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

Theorem [1, 3]
Verification of uninterpreted programs is undecidable.
Definition (Verification of Uninterpreted Programs)
Let $P \in \langle\text{stmt}\rangle$ be an uninterpreted program and let $\varphi$ be an assertion in the following grammar.

$$\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi$$

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

Theorem [1, 3]
Verification of uninterpreted programs is undecidable.
Coherence
How do we verify a single execution?

--- Execution $\rho$ ---

```plaintext
assume(T \neq F)  
b := F  
assume(x \neq y)  
d := key(x)  
assume(d = k)  
b := T  
r := x  
x := n(x)  
assume(x = y)
```

$\varphi \equiv b = T \Rightarrow key(r) = k$
How do we verify a single execution?

--- Execution $\rho$ ---

\begin{align*}
\text{assume}(T \neq F) & \\
\text{b} := F & \\
\text{assume}(x \neq y) & \\
\text{d} := \text{key}(x) & \\
\text{assume}(d = k) & \\
\text{b} := T & \\
\text{r} := x & \\
\text{x} := \text{n}(x) & \\
\text{assume}(x = y) & \\
\phi \equiv \text{b=}T \Rightarrow \text{key}(r)=k
\end{align*}

--- $\text{VC}(\rho, \phi)$ ---

\begin{align*}
T & \neq F \\
\land & \quad b_1 = F \\
\land & \quad x_0 \neq y_0 \\
\land & \quad d_1 = \text{key}(x_0) \\
\land & \quad d_1 = k_0 \\
\land & \quad b_2 = T \\
\land & \quad r_1 = x_0 \\
\land & \quad x_1 = \text{n}(x_0) \\
\land & \quad x_1 = y_0 \\
\Rightarrow & \quad (b_2 = T \Rightarrow \text{key}(r_1) = k_0)
\end{align*}
How do we verify a single execution?

<table>
<thead>
<tr>
<th>Execution ( \rho )</th>
<th>( VC(\rho, \varphi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assume(T \neq F)</code></td>
<td>( T \neq F )</td>
</tr>
<tr>
<td><code>b := F</code></td>
<td>( b_1 = F )</td>
</tr>
<tr>
<td><code>assume(x \neq y)</code></td>
<td>( x_0 \neq y_0 )</td>
</tr>
<tr>
<td><code>d := \text{key}(x)</code></td>
<td>( d_1 = \text{key}(x_0) )</td>
</tr>
<tr>
<td><code>assume(d = k)</code></td>
<td>( d_1 = k_0 )</td>
</tr>
<tr>
<td><code>b := T</code></td>
<td>( b_2 = T )</td>
</tr>
<tr>
<td><code>r := x</code></td>
<td>( r_1 = x_0 )</td>
</tr>
<tr>
<td><code>x := n(x)</code></td>
<td>( x_1 = n(x_0) )</td>
</tr>
<tr>
<td><code>assume(x = y)</code></td>
<td>( x_1 = y_0 )</td>
</tr>
<tr>
<td>( \varphi \equiv b=T \Rightarrow \text{key}(r)=k )</td>
<td>( (b_2 = T \Rightarrow \text{key}(r_1) = k_0) )</td>
</tr>
</tbody>
</table>

\( \varphi \) holds in every \( M \) in which \( \rho \) is feasible iff \( VC(\rho, \varphi) \) is valid in \( T_{\text{EUF}} \)
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 $\phi$ 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.
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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 $\varphi$ 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 on Ground Terms

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

$$
\begin{align*}
    t_1 &= t_1' \\
    t_2 &= t_2' \\
    \vdots \\
    t_k &= t_k'
\end{align*}
\Rightarrow

f(t_1, t_2, \ldots, t_k) = f(t_1', t_2', \ldots, t_k')$$
Congruence on Ground Terms

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

$$
\begin{align*}
\frac{t_1 = t'_1}{f(t_1, t_2, \ldots, t_k)} &= \frac{t_2 = t'_2}{f(t_1, t_2, \ldots, t_k)} = \cdots = \frac{t_k = t'_k}{f(t_1, t_2, \ldots, t_k)}
\end{align*}
$$

Interpretation

In every FO structure $M$,

if $[t_1]_M = [t'_1]_M$, $[t_2]_M = [t'_2]_M$, $\ldots$, and $[t_k]_M = [t'_k]_M$

then $[f(t_1, t_2, \ldots, t_k)]_M = [f(t'_1, t'_2, \ldots, t'_k)]_M$
assume$(x = y)$ $\rightarrow$ $x_1 := f(x)$ $\rightarrow$ $y_1 := f(y)$
Initially

\[ \hat{x} = \hat{y} \]

**assume** \((x = y) \rightarrow x_1 := f(x) \rightarrow y_1 := f(y)\)
Congruence Closure on Executions

Initially

\[
\begin{align*}
\text{assume}(x = y) & \quad \rightarrow \quad x_1 := f(x) & \quad \rightarrow \quad y_1 := f(y)
\end{align*}
\]
Congruence Closure on Executions

Initially

\[ \text{assume}(x = y) \rightarrow x_1 := f(x) \rightarrow y_1 := f(y) \]
Congruence Closure on Executions

Initially

\[ \text{assume}(x = y) \rightarrow x_1 := f(x) \rightarrow y_1 := f(y) \]
Initially

\[ \text{assume}(x = y) \rightarrow x_1 := f(x) \rightarrow y_1 := f(y) \]

\[ \varphi : x_1 = y_1 \]
Congruence Closure on Executions

Initially

\[ \text{assume}(x = y) \rightarrow x_1 := f(x) \rightarrow y_1 := f(y) \]

\[ \varphi : x_1 = y_1 \]

\( \varphi \) holds after the execution.
Congruence Closure on Executions

$n$ times

\[ \text{assume}(x = y) \rightarrow x := f(x) \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow y := f(y) \]
Congruence Closure on Executions

\[ \text{assume}(x = y) \rightarrow x := f(x) \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow y := f(y) \]

\[
\begin{array}{c}
\hat{x} \\
\equiv \\
\hat{y}
\end{array}
\]
assumption $(x = y) \rightarrow x := f(x) \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow y := f(y)$
Congruence Closure on Executions

\[ \text{assume}(x = y) \rightarrow x := f(x) \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow y := f(y) \]

\[ n \text{ times} \]

\[ \hat{x} \quad \hat{y} \]

\[ f(\hat{x}) \quad f^n(\hat{x}) \]

\[ \hat{y} \quad f(\hat{y}) \quad f^n(\hat{y}) \]
assume \((x = y)\) → \(x := f(x)\) → \(x := f(x)\) → \(y := f(y)\) → \(y := f(y)\)
assume\((x = y) \rightarrow x := f(x) \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow y := f(y)\)

\(\psi : x = y\)

\(\psi\) holds after the execution
Congruence Closure on Executions

Unbounded memory required to infer equality relationships in a streaming setting.
Congruence Closure on Executions

\[ n \text{ times } \]
\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]
Congruence Closure on Executions

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]
Congruence Closure on Executions

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]
Congruence Closure on Executions

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]

\[ \varphi : x_1 = y_1 \]
Congruence Closure on Executions

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]

\[ \varphi : x_1 = y_1 \]

\varphi \text{ holds after the execution}
Congruence Closure on Executions

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]

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

\[
\begin{align*}
\text{Term}(\epsilon, x) & = \widehat{x} & \text{for every } x \in V \\
\text{Term}(\rho \cdot "x := y", z) & = \text{Term}(\rho, y) & \text{if } z \text{ is } x \\
\text{Term}(\rho \cdot "x := f(z_1, \ldots)", y) & = f(\text{Term}(\rho, z_1), \ldots) & \text{if } y \text{ is } x \\
\text{Term}(\rho \cdot a, x) & = \text{Term}(\rho, x) & \text{otherwise}
\end{align*}
\]
Algebraic View of Executions

Terms Computed

$$\text{Term}(\epsilon, x) = \hat{x}$$
for every $$x \in V$$

$$\text{Term}(\rho \cdot "x := y", z) = \text{Term}(\rho, y)$$
if $$z$$ is $$x$$

$$\text{Term}(\rho \cdot "x := f(z_1, \ldots)", y) = f(\text{Term}(\rho, z_1), \ldots)$$
if $$y$$ is $$x$$

$$\text{Term}(\rho \cdot a, x) = \text{Term}(\rho, x)$$
otherwise

Equalities

$$\alpha(\epsilon) = \emptyset$$

$$\alpha(\rho \cdot \text{assume}(x = y)) = \alpha(\rho) \cup \{(\text{Term}(\rho, x), \text{Term}(\rho, y))\}$$

$$\alpha(\rho \cdot a) = \alpha(\rho)$$
otherwise
Terms Computed

\[
\begin{align*}
\text{Term}(\epsilon, x) &= \hat{x} & \text{for every } x \in V \\
\text{Term}(\rho \cdot "x := y", z) &= \text{Term}(\rho, y) & \text{if } z \text{ is } x \\
\text{Term}(\rho \cdot "x := f(z_1, \ldots)", y) &= f(\text{Term}(\rho, z_1), \ldots) & \text{if } y \text{ is } x \\
\text{Term}(\rho \cdot a, x) &= \text{Term}(\rho, x) & \text{otherwise}
\end{align*}
\]

Equalities

\[
\begin{align*}
\alpha(\epsilon) &= \emptyset \\
\alpha(\rho \cdot "\text{assume}(x = y)") &= \alpha(\rho) \cup \{(\text{Term}(\rho, x), \text{Term}(\rho, y))\} \\
\alpha(\rho \cdot a) &= \alpha(\rho) & \text{otherwise}
\end{align*}
\]

Disequalities

\[
\begin{align*}
\beta(\epsilon) &= \emptyset \\
\beta(\rho \cdot "\text{assume}(x \neq y)") &= \beta(\rho) \cup \{(\text{Term}(\rho, x), \text{Term}(\rho, y))\} \\
\beta(\rho \cdot a) &= \beta(\rho) & \text{otherwise}
\end{align*}
\]
An execution is **coherent** if it is **memoizing** and has **early assumes**.

\[ \text{Coherence} = \text{Memoizing} + \text{Early Assumes} \]
An execution is coherent if it is memoizing and has early assumes.
Coherence: Memoizing

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

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

we have the following.

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

Here,

- $\text{ComputedTerms}(\sigma) = \{\text{Term}(\pi, v) \mid v \in V, \pi \text{ is a prefix of } \sigma\}$,
- $\equiv_{\alpha(\rho)}$ is the smallest congruence induced by $\alpha(\rho)$. 
assume \((T \neq F)\);
\[ b := F; \]
\[ \textbf{while} (x \neq y) \{ \]
\[ \quad d := \text{key}(x); \]
\[ \quad \textbf{if} (d = k) \textbf{then} \{ \]
\[ \quad \quad b := T; \]
\[ \quad \quad r := x; \]
\[ \quad \}\]
\[ x := n(x); \]
\[ \}\]

- All executions of this program are \textit{vacuously memoizing}.
- No term is recomputed.
**Example execution: Non Memoizing**

\[ \text{assume}(x = y) \rightarrow x := f(x) \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow y := f(y) \]

\[ \hat{x} = \hat{y} \]

Re-computation of terms deemed equivalent by \( \hat{x} = \hat{y} \).

The older term \( f(\hat{x}) \) has been dropped.

**NOT a memoizing execution**
Example execution: Memoizing

\[\begin{align*}
\text{assume}(x = y) & \rightarrow x := f(x) \rightarrow y := f(y) \rightarrow \ldots \rightarrow x := f(x) \rightarrow y := f(y) \\
\end{align*}\]

\(n\) times

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 \( \rho \) is said to have early assumes if for every prefix of \( \rho \) of the form
\[
\sigma' = \sigma \cdot \text{"assume}(x = y)"
\]
we have the following.

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

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 execution: Violation of Early Assumes

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]

\[ n \text{ times} \]

\[ x_1 := f(x) \rightarrow y_1 := f(y) \rightarrow \cdots \rightarrow x_1 := f(x) \rightarrow y_1 := f(y_1) \rightarrow \text{assume}(x = y) \]

Superterms of \( \hat{x} \) and \( \hat{y} \) dropped before equality assume.

Does **NOT** satisfy early assumes.
Example execution: Early Assumes

\[\text{assume}(x = y) \rightarrow x := f(x) \rightarrow y := f(y) \quad \text{\(n\) times} \rightarrow x := f(x) \rightarrow y := f(y)\]

✓ Early Assume
Coherence

\begin{itemize}
\item In every execution, equality assume \textbf{assume}(x = y) occurs on terms without any superterms.
\item All executions are coherent!
\end{itemize}

\begin{verbatim}
assume (T \neq F);
b := F;
while (x \neq y) {
d := key(x);
if (d = k) then {
b := T;
r := x;
}
x := n(x);
}
\end{verbatim}
An uninterpreted program $P \in \langle \text{stmt} \rangle$ is coherent if all executions of $P$ are coherent.
An uninterpreted program $P \in \langle \text{stmt} \rangle$ 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 $\rho$,

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

- The question $\text{Exec}(P) \subseteq L_{\varphi}^{\text{coherent}}$ is decidable.
Regularity of Feasible Coherent Executions

- $P \models \varphi$ iff $P \neg \varphi \models false$, where $P \neg \varphi = P; \text{assume}(\neg \varphi)$
- Regular language $L_{coh-feas}$ such that for any coherent execution $\rho$,
  
  $$\rho \in L_{coh-feas} \text{ iff } \rho \text{ is feasible in some FO-structure } M$$

- $P \models \varphi$ iff $\text{Exec}(P \neg \varphi) \cap L_{coh-feas} = \emptyset$
Streaming Congruence Closure

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

  \[
  (\sim, d, F)
  \]

  Equivalence on variables

  \[
  \begin{align*}
  [x_1, x_3] \\
  [x_2, x_4, x_5] \\
  \vdots \\
  [x_1, x_3] \neq [x_6] \\
  \vdots
  \end{align*}
  \]

  Disequalities b/w eq. classes

  Partial func. relationships b/w eq. classes

  \[
  f([x_1, x_3]) = [x_2, x_4, x_5] \\
  \vdots
  \]
Transitions $\delta$ update these relationships in a streaming fashion.
Transitions $\delta$ update these relationships in a streaming fashion.

$x_1 = f(x)$
Transitions $\delta$ update these relationships in a streaming fashion.

$$x_1 = f(x) \rightarrow y_1 = f(y)$$
Transitions $\delta$ update these relationships in a streaming fashion.

\[ x_1 = f(x) \rightarrow y_1 = f(y) \rightarrow \text{assume}(x = y) \]
Transitions $\delta$ update these relationships in a streaming fashion.

\[ x_1 = f(x) \rightarrow y_1 = f(y) \rightarrow \text{assume}(x = y) \rightarrow \text{assume}(x \neq y) \]
Correctness of $A_{\text{coh-feas}}$

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

- If $\rho$ is not feasible in any $M$, then $q = q_{\text{reject}}$
- Otherwise, $q = (\sim, d, P)$ with
  - $\text{Term}(\rho, x) \equiv_{\alpha(\rho)} \text{Term}(\rho, y)$ iff $[x]_\sim = [y]_\sim$.
  - $([x]_\sim, [y]_\sim) \in d$ iff there is $(t_x, t_y) \in \beta(\rho)$ such that $t_x \equiv_{\alpha(\rho)} \text{Term}(\rho, x)$ and $t_y \equiv_{\alpha(\rho)} \text{Term}(\rho, y)$.
  - $f(\text{Term}(\rho, x)) \equiv_{\alpha(\rho)} \text{Term}(\rho, y)$ iff $F(f)([x]_\sim) = [y]_\sim$
Decidability of Checking Coherence [1]

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

$$\rho \in L(A_{\text{check-coh}}) \text{ iff } \rho \text{ is coherent}$$
Decidability of Checking Coherence [1]

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

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

- $A_{\text{check-coh}}$ ignores all letters of the form "assume($x \neq y$)".
- States of $A_{\text{check-coh}}$ maintain ($\sim$, $F$, $B$):
  - $\sim$ and $F$ are as in $A_{\text{coh-feas}}$
  - $B$ 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
\begin{itemize}
  \item \textbf{assume} \ (x \neq z); \\
  y := n(x); \\
  \textbf{assume} \ (y \neq z); \\
  y := n(y); \\
  \textbf{while} \ (y \neq z) \ { \\
    x := n(x); \\
    } \\
  y := n(y); \\
  } \\
  \phi \equiv \ z = n(n(x))
\end{itemize}
**k-Coherence**

```plaintext
assume (x \neq z);
y := n(x);
assume (y \neq z);
y := n(y);
while (y \neq z) {
    x := n(x);
    y := n(y);
}

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

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

NOT coherent
\textbf{k-Coherence}

\begin{verbatim}
assume (x \neq z);
y := n(x);
assume (y \neq z);
g := y;
y := n(y);
while (y \neq z) {
x := n(x);
g := y;
y := n(y);
}
\varphi \equiv z = n(n(x))
\end{verbatim}

\begin{itemize}
\item Can be \textit{made coherent}.
\item By adding additional \textit{ghost variables} and assignments to them.
\item Write-only and do not change semantics.
\end{itemize}
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 ($V \cap G = \emptyset$).

Let $\Pi_G = \Pi \cup \{"g := x" \mid g \in G, x \in V\}$.

An execution over $V$ is $k$-coherent if there is an execution $\rho'$ over $\Pi_G$ such that $\rho'$ is coherent and $\rho' \downharpoonright \Pi = \rho$.

A programs is $k$-coherent if all its executions are.
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 ($V \cap G = \emptyset$).
Let $\Pi_G = \Pi \cup \{“g := x” \mid g \in G, x \in V\}$.
An execution over $V$ is $k$-coherent if there is an execution $\rho'$ over $\Pi_G$ such that $\rho'$ is coherent and $\rho'|_{\Pi} = \rho$.
A program 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

\[
\text{assume } (T \neq F);
\]

\[
\text{if } (a \leq b) \text{ then } \{
\]

\[
\text{if } (a \leq c) \text{ then }
\]

\[
\text{min} := a;
\]

\[
\text{else } \text{min} := c;
\]

\[
\}
\]

\[
\text{else } \{
\]

\[
\text{if } (b \leq c) \text{ then }
\]

\[
\text{min} := b;
\]

\[
\text{else } \text{min} := c;
\]

\[
\}
\]

\[
\varphi \equiv \text{min} \leq a \land \text{min} \leq b \\
\land \text{min} \leq c
\]
Adding Interpretations

\[
\begin{align*}
\text{assume} \ (T \neq F); \\
\text{if} \ (a \leq b) \text{ then } \{ \\
\quad \text{if} \ (a \leq c) \text{ then} \\
\quad \quad \min := a; \\
\quad \text{else} \ \min := c; \\
\} \\
\text{else} \{ \\
\quad \text{if} \ (b \leq c) \text{ then} \\
\quad \quad \min := b; \\
\quad \text{else} \ \min := c; \\
\}
\end{align*}
\]

\[\varphi \equiv \min \leq a \land \min \leq b \land \min \leq c\]

Find the minimum of \(a\), \(b\) and \(c\)

Does not hold in all \(M\).
Adding Interpretations

\[
\text{assume } (T \neq F);
\]
\[
\text{if } (a \leq b) \text{ then } \{
\quad \text{if } (a \leq c) \text{ then }
\quad \quad \text{min} := a;
\quad \text{else } \text{min} := c;
\}
\]
\[
\text{else } \{
\quad \text{if } (b \leq c) \text{ then }
\quad \quad \text{min} := b;
\quad \text{else } \text{min} := c;
\}
\]
\[
\varphi \equiv \text{min} \leq a \land \text{min} \leq b \\
\quad \land \text{min} \leq c
\]

Find the minimum of \( a, b \) and \( c \)

Does not hold in all \( M \).
Adding Interpretations

```plaintext
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;
}

ϕ ≡ min ≤ a ∧ min ≤ b ∧ min ≤ c
```

Find the minimum of \( a, b \) and \( c \)

This program satisfies \( ϕ \) if \( ≤ \) is interpreted as a total order:

- \( ∀x \cdot x ≤ x \)
- \( ∀x, y, z \cdot x ≤ y ∧ y ≤ z \implies x ≤ z \)
- \( ∀x, y \cdot x ≤ y ∧ y ≤ x \implies x = y \)
Definition (Verification Modulo Axioms)

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

$$\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi$$
Definition (Verification Modulo Axioms)

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

\[
\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi
\]

$P \models \varphi$ modulo $A$
Adding Interpretations

Definition (Verification Modulo Axioms)
Let $P \in \langle \text{stmt} \rangle$ be an uninterpreted program over vocabulary $\Sigma$. Let $A$ be a set of first order sentences over $\Sigma$ and let $\varphi$ be an assertion in the following grammar.

$$\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi$$

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

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

$$
\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi
$$

$P \models \varphi$ modulo $A$ iff for every execution $\rho \in \text{Exec}(P)$ and for every FO structure $M$ such that $M \models A$ and $\rho$ is feasible in $M$,
**Definition (Verification Modulo Axioms)**

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

$$\varphi ::= \text{true} \mid x = y \mid R(z) \mid \varphi \lor \varphi \mid \neg \varphi$$

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


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

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

\[ x_1 := f(x, y) \quad \rightarrow \quad y_1 := f(y, x) \]
Example

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

\[ x_1 := f(x, y) \quad \rightarrow \quad y_1 := f(y, x) \]

re-computation modulo \( A \)
Example

\[ A = \{ \forall x, y. f(x, y) = f(y, x) \} \]

\[ x_1 := f(x, y) \rightarrow y_1 := f(y, x) \]

re-computation modulo \( A \)

\[ x_1 := f(x, y) \rightarrow y_1 := f(y, x') \rightarrow z := g(x_1) \rightarrow z' := g(y_1) \rightarrow \text{assume}(x = x') \]
Example

\[ A = \{ \forall x, y. f(x, y) = f(y, x) \} \]

\[ x_1 := f(x, y) \rightarrow y_1 := f(y, x) \]

\[ x_1 := f(x, y) \rightarrow y_1 := f(y, x') \rightarrow z := g(x_1) \rightarrow z' := g(y_1) \rightarrow \text{assume}(x = x') \]

Implied equality
\[ z = z' \]

re-computation modulo \( A \)
Definition (Memoizing modulo axioms)
Let $A$ be a set of axioms and let $\rho \in \Pi^*$ be an execution. Then, $\rho$ is said to be memoizing modulo $A$ if the following holds.
Let $\sigma' = \sigma \cdot "x:=f(z)"$ be a prefix of $\rho$. If there is a term $t' \in \text{ComputedTerms}(\sigma)$ such that $t' \equiv_{A \cup \kappa(\sigma)} \text{Term}(\sigma', x)$, then there must exist some variable $y \in V$ such that $\text{Term}(\sigma, y) \equiv_{A \cup \kappa(\sigma)} t$. 
Memoizing Modulo Axioms

Definition (Memoizing modulo axioms)
Let $A$ be a set of axioms and let $\rho \in \Pi^*$ be an execution. Then, $\rho$ is said to be memoizing modulo $A$ if the following holds.
Let $\sigma' = \sigma \cdot \text{"x := f(z)"}$ be a prefix of $\rho$. 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,

$$\kappa(\varepsilon) = \emptyset$$
$$\kappa(\rho \cdot \text{"assume}(x = y)"") = \kappa(\rho) \cup \{(\text{Term}(\rho, x) = \text{Term}(\rho, y))\}$$
$$\kappa(\rho \cdot \text{"assume}(x \neq y)"") = \kappa(\rho) \cup \{(\text{Term}(\rho, x) \neq \text{Term}(\rho, y))\}$$
$$\kappa(\rho \cdot \text{"R}(z_1, \ldots))" = \kappa(\rho) \cup \{R(\text{Term}(\rho, z_1), \ldots)\}$$
$$\kappa(\rho \cdot a) = \kappa(\rho) \quad \text{otherwise}$$
Definition (Early assumes modulo axioms)

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

Let $\sigma' = \sigma \cdot \text{“assume}(c)\text{”}$ be a prefix of $\rho$, where $c$ is any of $x = y$, $x \not= y$, $R(z)$, or $\neg R(z)$.

Let $t \in \text{ComputedTerms}(\sigma)$ be a term computed in $\sigma$ such that $t$ has been dropped, i.e., for every $x \in V$, we have $\text{Term}(\sigma, x) \not\equiv_{A \cup \kappa(\sigma)} t$.

For any term $t' \in \text{ComputedTerms}(\sigma)$, if $t \equiv_{A \cup \kappa(\sigma')} t'$, then $t \equiv_{A \cup \kappa(\sigma)} t'$. 


### Verification Modulo Axioms - Decidability Landscape [2]

<table>
<thead>
<tr>
<th>Relational axioms</th>
<th>Decidability</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPR</td>
<td>✗</td>
</tr>
<tr>
<td>Reflexivity</td>
<td>✓</td>
</tr>
<tr>
<td>Irreflexivity</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetry</td>
<td>✓</td>
</tr>
<tr>
<td>Transitivity</td>
<td>✓</td>
</tr>
<tr>
<td>Partial Order</td>
<td>✓</td>
</tr>
<tr>
<td>Total Order</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional axioms</th>
<th>Decidability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associativity</td>
<td>✗</td>
</tr>
<tr>
<td>Commutativity</td>
<td>✓</td>
</tr>
<tr>
<td>Idempotence</td>
<td>✓</td>
</tr>
</tbody>
</table>

### Combinations

<table>
<thead>
<tr>
<th>All combinations of decidable axioms</th>
<th>Decidability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Thank You!
Coherence Modulo Commutativity

Homomorphism $h_{\text{comm}}^f$ uses auxiliary variable $v^* \notin V$:

$$h_{\text{comm}}^f(a) = \begin{cases} 
  a \cdot "v^* := f(y, x)" \cdot "\text{assume}(z = v^*)" & \text{if } a = "z := f(x, y)"
  \\
  a & \text{otherwise}
\end{cases}$$

Coherence Modulo Commutativity

An execution $\rho$ is coherent modulo $A$ iff $h_{\text{comm}}^f(a)$ is coherent modulo $\emptyset$.

Feasibility Modulo Commutativity

An execution $\rho$ is feasible modulo $A$ iff $h_{\text{comm}}^f(a)$ is feasible modulo $\emptyset$. 
**Decidable verification of uninterpreted programs.**

**What’s decidable about program verification modulo axioms?**

M. Müller-Olm, O. Rüthing, and H. Seidl. 
**Checking herbrand equalities and beyond.**