Commutativity in Concurrent Program Verification

Dominik Klumpp
klumpp@informatik.uni-freiburg.de
University of Freiburg

joint work with: Azadeh Farzan (University of Toronto)
                Andreas Podelski (University of Freiburg)
                Marcel Ebbinghaus (University of Freiburg)

AVM 2022
Example Program

\[
\{ \ x = y = i = j = 0 \ \}
\]

\[
\begin{align*}
\text{while} \ (i < n) \ { &} \\
& \ x += A[i]; \\
& \ i++; \\
\text{\ } \\
\text{while} \ (j < n) \ { &} \\
& \ y += A[j]; \\
& \ j++; \\
\end{align*}
\]

\[
\{ \ x = y \ \}
\]
Naïve Sequentialization

Counterexample:
\[ \tau = (i < n) \land (x = \sum_{k=0}^{i} A[k]) \land (y = \sum_{k=0}^{j} A[k]) \land (i \leq n) \land (j \leq n) \land (x = y) \land (i = j) \land (i \geq n) \land (j \geq n) \]
Naïve Sequentialization

\[ x = \sum_{k=0}^{i} A[k] \land y = \sum_{k=0}^{j} A[k] \land i \leq n \land j \leq n \]
Many statements **commute**: execution order does not matter

**Example:** \( x + = A[i] \) \( y + = A[j] \) \( \sim \) \( y + = A[j] \) \( x + = A[i] \)

**Key Property:** Correct traces only equivalent to correct traces.

**Reduction:** One representative trace for each equivalence class.
Many statements **commute**: execution order does not matter

**Example:** \[ x+=A[i] \quad y+=A[j] \sim y+=A[j] \quad x+=A[i] \]

\[\Rightarrow\] equivalence between program interleavings
Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

**Example:** \( x+=A[i] \) \( y+=A[j] \) \( \sim \) \( y+=A[j] \) \( x+=A[i] \)

\( \Rightarrow \) equivalence between program interleavings

Extension: **proof-sensitive** commutativity

**Example:** \( B[k]:=c \) commutes with \( B[l]:=d \) if proof guarantees \( k \neq l \lor c = d \)
Commutativity-Based Equivalence

Many statements commute: execution order does not matter

Example: \[ x+=A[i] \quad y+=A[j] \quad \sim \quad y+=A[j] \quad x+=A[i] \]

\[ \Rightarrow \text{equivalence between program interleavings} \]

Extension: proof-sensitive commutativity

Example: \( B[k] := c \) commutes with \( B[l] := d \) if proof guarantees \( k \neq l \lor c = d \)

Typical Cases: aliasing, conditional updates (CAS), blocking statements (locks)
Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

**Example:** \[x+=A[i] \quad y+=A[j] \sim y+=A[j] \quad x+=A[i]\]

\[\Rightarrow \text{equivalence between program interleavings}\]

**Extension:** **proof-sensitive** commutativity

**Example:** \[B[k]:=c \quad \text{commutes with} \quad B[l]:=d \quad \text{if proof guarantees} \quad k \neq l \lor c = d\]

**Typical Cases:** aliasing, conditional updates (CAS), blocking statements (locks)

**Key Property:** Correct traces only equivalent to correct traces.
Commutativity-Based Equivalence

Many statements **commute**: execution order does not matter

**Example:** \( x + = A[i] \) \( y + = A[j] \) \( \sim \) \( y + = A[j] \) \( x + = A[i] \)

\( \Rightarrow \) equivalence between program interleavings

**Extension:** proof-sensitive commutativity

**Example:** \( B[k] := c \) commutes with \( B[l] := d \) if proof guarantees \( k \neq l \vee c = d \)

**Typical Cases:** aliasing, conditional updates (CAS), blocking statements (locks)

**Key Property:** Correct traces only equivalent to correct traces.

**Reduction**

One representative trace for each equivalence class
Naïve Sequentialization

Counterexample:

\[ \tau = \begin{cases} 
  i < n & x = i \sum_{k=0}^{n} A[k] \\
  j < n & y = j \sum_{k=0}^{n} A[k] \\
  i \geq n & x = y \\
  j \geq n & x = y \\
 \end{cases} \]
Reduction I

Counterexample: 

\[ \tau = \begin{cases} i < n \\
\text{x} = i \sum_{k=0}^{n} A[k] \end{cases} \]
Algorithmic Verification of Reductions

Iteratively construct \textbf{Floyd/Hoare-style proof} of program
Iteratively construct **Floyd/Hoare-style proof** of program

**complex proofs** to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

⇝ reduction may have simpler proof

⇝ exponential proof checking to show that proof covers all interleavings

⇝ compactly represent reductions

[1] Ebbinghaus. Tight Integration of Partial Order Reduction into Trace Abstraction Refinement

Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

**complex proofs** to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

⇝ reduction may have **simpler proof**
Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

*complex proofs* to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

~⇒ reduction may have **simpler proof**

**exponential proof checking** to show that proof covers all interleavings
Algorithmic Verification of Reductions

Iteratively construct **Floyd/Hoare-style proof** of program

**complex proofs** to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

⇒ reduction may have **simpler proof**

**exponential proof checking** to show that proof covers all interleavings

⇒ **compactly represent** reductions
Iteratively construct **Floyd/Hoare-style proof** of program

**complex proofs** to cover *all* interleavings

- **qualitatively:** need quantified / nonlinear / ... assertions
- **quantitatively:** need many distinct proof assertions

⇝ reduction may have **simpler proof**

**exponential proof checking** to show that proof covers all interleavings

⇝ **compactly represent** reductions

Evaluation

Implemented in **ULTIMATE GEMCUTTER** software model checker
Evaluated on SV-COMP’21 benchmarks and benchmarks of **WEAVER** tool

analyzed *50* more programs using significantly less time (**≈ 50%**), memory (**≈ 27%**), and refinement rounds (**≈ 64%**).
Reduction: One representative trace for each equivalence class
**Reduction:** One representative trace for each equivalence class

\[ \text{red}^I_{\preceq}(P) \]
**Reduction**: One representative trace for each equivalence class

$$\text{red}_{I \leq}^I(P)$$

program to be verified
**Reduction:** One representative trace for each equivalence class

**commutativity relation** $I$

defines equivalence classes

$\text{red}^I(P)$

program to be verified
Reduction: One representative trace for each equivalence class

Commutativity relation $I$ defines equivalence classes

Preference order $\preceq$ selects representatives for each equivalence class

$\text{red}^I_{\preceq}(P)$

Program to be verified
Preference Orders

Selecting the right representatives

Reduction 1

Counterexample:

\[ \tau = \begin{align*}
  i &< n \\
x &+= A[i] \\
i &++ \\
\end{align*} \]

\[ \begin{align*}
  j &< n \\
y &+= A[j] \\
j &++ \\
\end{align*} \]

\[ \begin{align*}
  i &\geq n \\
  j &\geq n \\
\end{align*} \]

\[ \begin{align*}
  i &< n \\
x &+= A[i] \\
i &++ \\
\end{align*} \]

\[ \begin{align*}
  j &< n \\
y &+= A[j] \\
j &++ \\
\end{align*} \]

\[ \begin{align*}
  i &\geq n \\
  j &\geq n \\
\end{align*} \]

\[ \begin{align*}
  i &< n \\
x &+= A[i] \\
i &++ \\
\end{align*} \]

\[ \begin{align*}
  j &< n \\
y &+= A[j] \\
j &++ \\
\end{align*} \]

\[ \begin{align*}
  i &\geq n \\
  j &\geq n \\
\end{align*} \]
Reduction I

\[
x = \sum_{k=0}^{i} A[k] \land i \leq n \land y = 0 \land j = 0
\]

\[
x = \sum_{k=0}^{n} A[k] \land y = \sum_{k=0}^{j} A[k] \land j \leq n
\]

Counterexample:

\[
t = \sum_{k=0}^{i} A[k] \land i \leq n \land y = 0 \land j = 0
\]

\[
x = i \quad \sum_{k=0}^{j} A[k] \land i \leq n \land j \leq n
\]

\[
x = y \land i = j \land x = y \land j = n \land i \geq n
\]

\[
x = i \quad \sum_{k=0}^{j} A[k] \land i \leq n \land j = 0 \land y = 0
\]

\[
x = n \quad \sum_{k=0}^{j} A[k] \land y = j \quad \sum_{k=0}^{j} A[k] \land j \leq n
\]

\[
j \geq n
\]

\[
x = y \land j \geq n
\]
Counterexample:

\[ \tau = i < n \quad x += A[i] \quad j < n \quad y += A[j] \quad i++ \quad j++ \quad i >= n \quad j >= n \]

\[ x = i \sum_{k=0}^{n} A[k] \quad y = j \sum_{k=0}^{n} A[k] \quad i <= n \quad j <= n \quad x = y \quad i = j \]

\[ x = y \quad j >= n \quad x = y \quad i >= n \]

\[ x = i \sum_{k=0}^{n} A[k] \quad i <= n \quad y = 0 \quad j = 0 \quad i >= n \quad y = j \sum_{k=0}^{n} A[k] \quad j <= n \]
Reduction II

Counterexample:

\[ x = y \land i \geq n \]

\[ x = y \land i = j \]

\[ x = y \land j \geq n \]

\[ i = \sum_{k=0}^{i} A[k] \land y = j \sum_{k=0}^{j} A[k] \land i \leq n \land j \leq n \]

\[ x = y \land i = j \land j \geq n \]

\[ x = y \land i = j \land j \geq n \]

\[ i = \sum_{k=0}^{i} A[k] \land j = \sum_{k=0}^{j} A[k] \land i \leq n \land j = 0 \land x = 0 \]

\[ j = \sum_{k=0}^{j} A[k] \land i = \sum_{k=0}^{i} A[k] \land j \leq n \land j = 0 \]

\[ i = \sum_{k=0}^{i} A[k] \land j = \sum_{k=0}^{j} A[k] \land i \leq n \land j = 0 \land x = 0 \]

\[ j = \sum_{k=0}^{j} A[k] \land i = \sum_{k=0}^{i} A[k] \land j \leq n \land j = 0 \land x = 0 \]
Preference orders characterize choice of reduction
Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
- keep only **most preferred representative** per equivalence class

\[
red^I_{\preceq}(P) := \{ \min_{\preceq}[\tau]_{\sim_I} \mid \tau \in P \}
\]
Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
- keep only most preferred representative per equivalence class

\[ red_I(P) := \{ \min_{\preceq} [\tau]_{\sim_I} \mid \tau \in P \} \]

- independent of commutativity
Preference orders characterize choice of reduction

- order interleavings from most preferred (smallest) to least preferred (greatest)
- keep only most preferred representative per equivalence class

\[ \text{red}_I(P) := \{ \min_{\preceq}[\tau]_{\sim_I} \mid \tau \in P \} \]

- independent of commutativity
- same scheme of preference order applies to different programs
Algorithmic construction of reductions using techniques from partial order reduction:
Algorithmic construction of reductions using techniques from partial order reduction:

- finite representation as control flow graphs

→ More on preference orders in Marcel’s talk
Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- finite representation as control flow graphs
  - constructed using variant of sleep set technique

More on preference orders in Marcel’s talk
Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings**: proofs not unnecessarily complex
Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of sleep set technique
- **no redundant interleavings**: proofs not unnecessarily complex
- **compact** representation

More on preference orders in Marcel's talk
Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of sleep set technique
- **no redundant interleavings**: proofs not unnecessarily complex
- **compact** representation
  - through weakly persistent membranes
Positional Lexicographic Preference Orders

Algorithmic construction of reductions using techniques from partial order reduction:

▶ finite representation as control flow graphs
  • constructed using variant of sleep set technique
▶ no redundant interleavings: proofs not unnecessarily complex
▶ compact representation
  • through weakly persistent membranes
▶ linear-size representation in the best case
Algorithmic construction of reductions using techniques from partial order reduction:

- **finite representation** as control flow graphs
  - constructed using variant of **sleep set** technique
- **no redundant interleavings**: proofs not unnecessarily complex
- **compact** representation
  - through **weakly persistent membranes**
- **linear-size** representation in the best case

→ More on preference orders in Marcel’s talk
Commutativity Relations

at different abstraction levels

[work in progress; presented at Commute workshop @ PLDI’22]
Commutativity

Statements $s_1$ and $s_2$ commute

iff
Commutativity

Statements $s_1$ and $s_2$ commute

iff

the order of execution does not matter
Commutativity

Statements $s_1$ and $s_2$ commute

iff

the order of execution does not matter

($s_1s_2$ behaves exactly like $s_2s_1$)

Formally: $[s_1s_2] = [s_2s_1]$
Commutativity

Statements $s_1$ and $s_2$ commute

iff

the order of execution does not matter

$(s_1s_2 \text{ behaves exactly like } s_2s_1)$

for all programs and wrt. all properties

Formally: $[s_1s_2] = [s_2s_1]$
Commutativity

Statements $s_1$ and $s_2$ commute

iff

the order of execution does not matter

($s_1 s_2$ behaves similar enough to $s_2 s_1$)

for a given program and property
Commutativity

Statements \( s_1 \) and \( s_2 \) commute iff the order of execution does not matter (\( s_1 s_2 \) behaves similar enough to \( s_2 s_1 \)) for a given program and property.
Commutativity

Statements $s_1$ and $s_2$ commute iff
the order of execution does not matter
($s_1 s_2$ behaves similar enough to $s_2 s_1$)
for a given program and property
Statements $s_1$ and $s_2$ commute

iff

the order of execution does not matter

($s_1s_2$ behaves similar enough to $s_2s_1$)

for a given (partial) proof
Let $\Pi$ be a proof (a set of Hoare triples).

\[ \text{red}_I^I(P) \subseteq \mathcal{L}(\Pi) \]

$P$ is correct
Safe Commutativity

Let \( \Pi \) be a \textbf{proof} (a set of Hoare triples).

\[
\text{traces proven correct by } \Pi
\]

\[
\text{red}_{\geq}^{I}(P) \subseteq \mathcal{L}(\Pi)
\]

\[
P \text{ is correct}
\]
Safe Commutativity

Let $\Pi$ be a proof (a set of Hoare triples).

$\text{traces proven correct by } \Pi$

\[
\text{red}^I_{\geq} (P) \subseteq \mathcal{L}(\Pi) \quad I \text{ safe wrt. } \Pi
\]

$P$ is correct
Let $\Pi$ be a proof (a set of Hoare triples).

\[
\text{red}_{\geq}^I(P) \subseteq \mathcal{L}(\Pi) \quad I \text{ safe wrt. } \Pi \\
P \text{ is correct}
\]
Let $\Pi$ be a proof (a set of Hoare triples).

\[ \text{traces proven correct by } \Pi \subseteq \mathcal{L}(\Pi) \quad I \text{ safe wrt. } \Pi \]

\[ P \text{ is correct} \]

- commutativity $I_C$ based on (concrete) semantics: safe wrt. all proofs $\Pi$
Safe Commutativity

Let $\Pi$ be a proof (a set of Hoare triples).

\[ \text{traces proven correct by } \Pi \subseteq \text{correct traces} \]

\[ \text{traces in } \mathcal{L}(\Pi) \text{ only equivalent to correct traces} \]

\[ \text{commutativity } I_C \text{ based on (concrete) semantics: safe wrt. all proofs } \Pi \]

\[ \text{How to get safe commutativity for a particular proof } \Pi? \]
Safe Abstraction

Let $\alpha : Stmt \rightarrow Stmt$. 

Theorem (Safety)

If $\alpha$ satisfies $\triangleright$ abstraction: $J st K \subseteq J \alpha (st) K$ for all $st$, $\triangleright$ preservation: $\{ \phi \} \alpha (st) \{ \psi \}$ is valid, for all $\{ \phi \} st \{ \psi \} \in \Pi$ then $I \alpha$ is safe wrt. $\Pi$. 

Safe Abstraction

Let $\alpha : Stmt \rightarrow Stmt$.

$$I_\alpha := \{ (s_1, s_2) \mid \llbracket \alpha(s_1)\alpha(s_2) \rrbracket = \llbracket \alpha(s_2)\alpha(s_1) \rrbracket \}$$
Safe Abstraction

Let $\alpha : Stmt \rightarrow Stmt$.

$$I_\alpha := \{ (s_1, s_2) \mid \llbracket \alpha(s_1)\alpha(s_2) \rrbracket = \llbracket \alpha(s_2)\alpha(s_1) \rrbracket \}$$

**Theorem (Safety)**

If $\alpha$ satisfies

- **abstraction:** $\llbracket s \rrbracket \subseteq \llbracket \alpha(s) \rrbracket$ for all $s$
- **preservation:** $\{\phi\} s \{\psi\}$ is valid, for all $\{\phi\} s \{\psi\} \in \Pi$

then $I_\alpha$ is safe wrt. $\Pi$. 

Idea: Variable $x$ does not occur in the proof $\Rightarrow$ Ignore $x$ when determining commutativity
Instance: Projection to the Proof

Idea: Variable $x$ does not occur in the proof \(\Rightarrow\) Ignore $x$ when determining commutativity

Abstraction:
- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

Example: Let $\Pi = \{\{\top\} \ y:=x+x\ \{y \neq 1\}\}$. Then $\alpha_{\Pi}(y:=x+x)$: "assign $y$ to some even value (nondet.)"

$\alpha_{\Pi}(x:=0)$: "do not change $y"$

Now: $\alpha_{\Pi}(y:=x+x)$ commutes with $\alpha_{\Pi}(x:=0)$. 
Idea: Variable $x$ does not occur in the proof $\Rightarrow$ Ignore $x$ when determining commutativity

Abstraction:

- reads of irrelevant variables $\leadsto$ nondeterministic values
- assignment to irrelevant variables $\leadsto$ nondeterministic assignment (havoc)

Example

Let $\Pi = \{ \top \} \ y := x + x \ {y \neq 1} \}$. Then
**Idea:** Variable $x$ does not occur in the proof $\Rightarrow$ Ignore $x$ when determining commutativity

**Abstraction:**
- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

**Example**

Let $\Pi = \{ \{\top\} \ y:=x+x \ {y \neq 1} \}$. Then

$$\alpha_\Pi(y:=x+x) : \text{“assign } y \text{ to some even value (nondet.)”}$$
**Idea:** Variable $x$ does not occur in the proof $\Rightarrow$ Ignore $x$ when determining commutativity

**Abstraction:**
- reads of irrelevant variables $\rightsquigarrow$ nondeterministic values
- assignment to irrelevant variables $\rightsquigarrow$ nondeterministic assignment (havoc)

**Example**
Let $\Pi = \{ \top \} \{ y := x + x \} \{ y \neq 1 \}$. Then

\[
\alpha_\Pi(y := x + x) : \text{“assign } y \text{ to some even value (nondet.)”}
\]
\[
\alpha_\Pi(x := 0) : \text{“do not change } y \text{”}
\]
Instance: Projection to the Proof

Idea: Variable $x$ does not occur in the proof $\Rightarrow$ Ignore $x$ when determining commutativity

Abstraction:
- reads of irrelevant variables $\leadsto$ nondeterministic values
- assignment to irrelevant variables $\leadsto$ nondeterministic assignment (havoc)

Example

Let $\Pi = \{ \{ \top \} \ y := x+x \{ y \neq 1 \} \}$. Then

\[ \alpha_\Pi( y := x+x) : \text{“assign } y \text{ to some even value (nondet.)”} \]
\[ \alpha_\Pi( x := 0) : \text{“do not change } y \text{”} \]

Now: $\alpha_\Pi( y := x+x)$ commutes with $\alpha_\Pi( x := 0)$. 
Proposition: Projection to the proof is safe (it satisfies abstraction and preservation).
**Proposition:** Projection to the proof is safe (it satisfies abstraction and preservation).

**Advantages:**

**Limitations:**
Proposition: Projection to the proof is safe (it satisfies abstraction and preservation).

Advantages: often allows additional commutativity

Limitations:
- theoretically: may lose commutativity
- practically: introduces quantifiers

Generally: abstract commutativity ⊉ concrete commutativity

Solution: combine abstract with concrete commutativity
**Proposition:** Projection to the proof is safe (it satisfies *abstraction* and *preservation*).

**Advantages:**
- often allows additional commutativity
- abstraction easy to compute

**Limitations:**
- theoretically: may lose commutativity
- practically: introduces quantifiers

Generally: abstract commutativity $\not\subseteq$ concrete commutativity

**Solution:** combine abstract with concrete commutativity
Proposition: Projection to the proof is safe (it satisfies abstraction and preservation).

Advantages:
- often allows additional commutativity
- abstraction easy to compute

Limitations:
- theoretically: may lose commutativity
Proposition: Projection to the proof is safe (it satisfies abstraction and preservation).

Advantages:
- often allows additional commutativity
- abstraction easy to compute

Limitations:
- theoretically: may lose commutativity
- practically: introduces quantifiers
Proposition: Projection to the proof is safe (it satisfies abstraction and preservation).

Advantages:
- often allows additional commutativity
- abstraction easy to compute

Limitations:
- theoretically: may lose commutativity
- practically: introduces quantifiers

Generally: abstract commutativity $\not\supset$ concrete commutativity
Proposition: Projection to the proof is safe (it satisfies abstraction and preservation).

Advantages:
- often allows additional commutativity
- abstraction easy to compute

Limitations:
- theoretically: may lose commutativity
- practically: introduces quantifiers

Generally: abstract commutativity $\nleq$ concrete commutativity

Solution: combine abstract with concrete commutativity
Observation: **Union** of safe commutativity relations may be unsafe!
Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** \( \top \)
- **postcondition:** \( z = 2 \)
- **proof** \( \Pi: \{ \top \} \xRightarrow{x:=1} \{ \top \} \xRightarrow{x==1;z:=1} \{ \top \} \xRightarrow{x==2;z:=2} \{ z = 2 \} \)
Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** $\top$
- **postcondition:** $z = 2$
- **proof** $\Pi$: $\{\top\} x:=1 \{\top\} x==1; z:=1 \{\top\} x==2; z:=2 \{z = 2\}$
Combining Commutativity Relations

**Observation:** Union of safe commutativity relations may be unsafe!

**Example:**

- **precondition:** $\top$
- **postcondition:** $z = 2$
- **proof $\Pi$:** $\{\top\} \xRightarrow{x:=1} \{\top\} \xRightarrow{x==1;z:=1} \{\top\} \xRightarrow{x==2;z:=2} \{z = 2\}$

\[
\begin{align*}
x &:= 1 & x &:= 1; z &:= 1 & x &:= 2; z &:= 2 & \sim_{IC} & x &:= 1 & x &:= 2; z &:= 2 & x &:= 1; z &:= 1
\end{align*}
\]
Observation: **Union** of safe commutativity relations may be unsafe!

Example:

- **precondition:** \( \top \)
- **postcondition:** \( z = 2 \)
- **proof** \( \Pi: \{ \top \} x:=1 \{ \top \} x==1;z:=1 \{ \top \} x==2;z:=2 \{ z = 2 \} \)

\[
\begin{align*}
\text{x:=1} & \quad \text{x==1;z:=1} & \quad \text{x==2;z:=2} & \quad \sim_{I_C} \quad \text{x:=1} & \quad \text{x==2;z:=2} & \quad \text{x==1;z:=1} \\
\sim_{I_\alpha} \text{x==2;z:=2} & \quad \text{x:=1} & \quad \text{x==1;z:=1}
\end{align*}
\]
Idea: Sequentially combine commutativity relations
Idea: Sequentially combine commutativity relations

(1) abstract

\[ \tau_1 \sim I_\alpha \]

proven

⇒

(2) concrete

\[ \tau_2 \sim I_C \]

\[ \tau_3 \]

correct
Combining Commutativity Relations

**Idea:** Sequentially combine commutativity relations

\[ \begin{align*}
\tau_1 \sim_{I_\alpha} & \uparrow \text{proven} \\
\tau_2 \sim_{I_C} & \Rightarrow \\
\tau_3 & \uparrow \text{correct}
\end{align*} \]

(1) abstract \quad (2) concrete

Combination through **new proof rule:**

\[
red_{\leq}^{I_\alpha,I_C} (P) \subseteq \mathcal{L}(\Pi) \quad I_\alpha \text{ safe wrt. } \Pi
\]

\[
P \text{ is correct}
\]
Combining Commutativity Relations

**Idea:** Sequentially combine commutativity relations

\[
\begin{align*}
\tau_1 \sim_{I_\alpha} I_\alpha \quad &\quad (1) \text{ abstract} \\
\tau_2 \sim_{I_C} I_C \quad &\quad (2) \text{ concrete} \\
\tau_3 \quad &\quad \uparrow \\
\text{proven} \quad &\quad \Rightarrow \\
\text{correct} \quad &\quad 
\end{align*}
\]

Combination through **new proof rule**:

\[
\text{red}_{\leq}^{I_1, \ldots, I_n} (P) \subseteq \mathcal{L}(\Pi) \quad I_1, \ldots, I_n \text{ safe wrt. } \Pi \\
P \text{ is correct} \\
\]

\[
I_1 \sqsupset \ldots \sqsupset I_n
\]

New partial order reduction algorithm for \(n\) commutativity relations

"more abstract than"
Combining Commutativity Relations

Idea: Sequentially combine commutativity relations

\[ \tau_1 \sim I_\alpha \tau_2 \sim I_C \tau_3 \]

(1) abstract \hspace{1cm} (2) concrete

proven \hspace{1cm} \Rightarrow \hspace{1cm} correct

Combination through new proof rule:

\[ \text{red}_{\leq}^{I_1,\ldots,I_n}(P) \subseteq \mathcal{L}(\Pi) \quad I_1,\ldots,I_n \text{ safe wrt. } \Pi \quad I_1 \supseteq \ldots \supseteq I_n \]

\[ P \text{ is correct} \]

New partial order reduction algorithm for \( n \) commutativity relations
Conclusion
In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow **efficient proof checking**.
In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow **efficient proof checking**.

**Preference Orders:** Selection of representatives in reduction

- influences both proof simplicity and proof check efficiency
- trade-off between both aspects
In algorithmic verification, **commutativity-based reductions** can **simplify proofs** and allow efficient proof checking.

**Preference Orders:** Selection of representatives in reduction
- influences both proof simplicity and proof check efficiency
- trade-off between both aspects

**Commutativity Relations:** Determines notion of equivalence
- automatically **computed** and **safe wrt. a proof**
- e.g. derived from **safe abstractions**
- new **proof rule** and **algorithm** combine commutativity relations
In algorithmic verification, commutativity-based reductions can simplify proofs and allow efficient proof checking.

Preference Orders: Selection of representatives in reduction
- influences both proof simplicity and proof check efficiency
- trade-off between both aspects

Commutativity Relations: Determines notion of equivalence
- automatically computed and safe wrt. a proof
- e.g. derived from safe abstractions
- new proof rule and algorithm combine commutativity relations

Questions?