CS228 : Logic for Computer Science

Module 4: Hoare Logic and Program Verification

Lecture 21: Proving Loops Correct: Invariants and Variants

Instructor: S. Akshay

IIT Bombay, India

Logistics

Recall: Quiz 2

- **Friday, Oct 23rd, 6pm**
- Venue: CC 101/103/105 (*to be confirmed, will be announced on Piazza*)
- Syllabus:
  - Module 3 (Predicate Logic/FOL): all of Lectures 12–18
  - Module 4 (Hoare logic): from Lecture 19 up to and including today's lecture (Thursday, Oct 15th)

Topics Covered till date

Module 1: Propositional Logic

1. Syntax
2. Semantics
3. Formal proof systems
4. Normal Forms

Module 2: The Problem of Satisfiability

1. Propositional Resolution
2. DPLL
3. CDCL
4. SAT tool demo

Topics Covered till date

Module 1: Propositional Logic

1. Syntax
2. Semantics
3. Formal proof systems
4. Normal Forms

Module 2: The Problem of Satisfiability

1. Propositional Resolution
2. DPLL
3. CDCL
4. SAT tool demo

Module 3: Predicate Logic

1. Syntax of predicate logic
2. Semantics of predicate logic
3. Natural deduction for FOL
4. Structures, power and limits of FOL
5. Normal forms, DPLL(T) and SMT

Topics Covered till date

Module 1: Propositional Logic

1. Syntax
2. Semantics
3. Formal proof systems
4. Normal Forms

Module 2: The Problem of Satisfiability

1. Propositional Resolution
2. DPLL
3. CDCL
4. SAT tool demo

Module 3: Predicate Logic

1. Syntax of predicate logic
2. Semantics of predicate logic
3. Natural deduction for FOL
4. Structures, power and limits of FOL
5. Normal forms, DPLL(T) and SMT

Module 4: Hoare Logic and Program Verification

1. Introduction to Hoare logic
2. Proof rules for partial correctness
3. Today: Loop invariants and variants

Recap: A proof calculus for program correctness

$$\frac{\{\varphi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}}\ \text{Seq. Comp.} \qquad \frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

Recap: A proof calculus for program correctness

$$\frac{\{\varphi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}}\ \text{Seq. Comp.}$$

$$\frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$$\frac{\vdash_{AR} \varphi \to \varphi' \qquad \{\varphi'\}\ C\ \{\psi'\} \qquad \vdash_{AR} \psi' \to \psi}{\{\varphi\}\ C\ \{\psi\}}\ \text{Implied}$$

Recap: A proof calculus for program correctness

$$\frac{\{\varphi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}}\ \text{Seq. Comp.} \qquad \frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment} \qquad \frac{\vdash_{AR} \varphi \to \varphi' \qquad \{\varphi'\}\ C\ \{\psi'\} \qquad \vdash_{AR} \psi' \to \psi}{\{\varphi\}\ C\ \{\psi\}}\ \text{Implied}$$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

Recap: A proof calculus for program correctness

$$\frac{\{\varphi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}}\ \text{Seq. Comp.} \qquad \frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment} \qquad \frac{\vdash_{AR} \varphi \to \varphi' \qquad \{\varphi'\}\ C\ \{\psi'\} \qquad \vdash_{AR} \psi' \to \psi}{\{\varphi\}\ C\ \{\psi\}}\ \text{Implied}$$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

We say sequent $\vdash_{par} \{\varphi\}\ P\ \{\psi\}$ is valid if ψ can be derived from φ by applying these rules!

Recap: A proof calculus for program correctness

$$\frac{\{\varphi\}\ C_1\ \{\eta\} \qquad \{\eta\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1; C_2\ \{\psi\}}\ \text{Seq. Comp.} \qquad \frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment} \qquad \frac{\vdash_{AR} \varphi \to \varphi' \qquad \{\varphi'\}\ C\ \{\psi'\} \qquad \vdash_{AR} \psi' \to \psi}{\{\varphi\}\ C\ \{\psi\}}\ \text{Implied}$$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

We say sequent $\vdash_{par} \{\varphi\}\ P\ \{\psi\}$ is valid if ψ can be derived from φ by applying these rules!

This proof system is sound so we can use this to infer partial correctness, i.e., $\models_{par}$.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $\boxed{t := x;\ x := y;\ y := t}$, which is meant to swap x and y.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$t := x;$

$x := y;$

$y := t;$

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$t := x;$

$x := y;$

$y := t;$

$\{x = y_0 \wedge y = x_0\}$

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$t := x;$

$x := y;$

$\{x = y_0 \wedge t = x_0\}$

$y := t;$

$\{x = y_0 \wedge y = x_0\}$ Assignment

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$t := x;$

$\{y = y_0 \wedge t = x_0\}$

$x := y;$

$\{x = y_0 \wedge t = x_0\}$ Assignment

$y := t;$

$\{x = y_0 \wedge y = x_0\}$ Assignment

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$\{y = y_0 \wedge x = x_0\}$

$t := x;$

$\{y = y_0 \wedge t = x_0\}$ Assignment

$x := y;$

$\{x = y_0 \wedge t = x_0\}$ Assignment

$y := t;$

$\{x = y_0 \wedge y = x_0\}$ Assignment

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$\{y = y_0 \wedge x = x_0\}$ Implied

$t := x;$

$\{y = y_0 \wedge t = x_0\}$ Assignment

$x := y;$

$\{x = y_0 \wedge t = x_0\}$ Assignment

$y := t;$

$\{x = y_0 \wedge y = x_0\}$ Assignment

Recall: Assignment rule

$$\frac{}{\{\psi[E/x]\}\ x := E\ \{\psi\}}\ \text{Assignment}$$

$\psi[E/x]$: replace every *free* occurrence of x in ψ by E.

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$\{y = y_0 \wedge x = x_0\}$ Implied

$t := x;$

$\{y = y_0 \wedge t = x_0\}$ Assignment

$x := y;$

$\{x = y_0 \wedge t = x_0\}$ Assignment

$y := t;$

$\{x = y_0 \wedge y = x_0\}$ Assignment

- x_0 and y_0 are logical variables: they remember the initial values (recall **Fac**$_2$)!

The dreaded 8am Pop quiz!

Consider program **Swap** defined as $t := x;\ x := y;\ y := t$, which is meant to swap x and y.

Look at the Hoare triple $\{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

1. What are the pre and post conditions of this Hoare triple?
2. What are the program variables and what are the logical variables?
3. Show that this Hoare triple is valid, i.e., $\vdash_{par} \{x = x_0 \wedge y = y_0\}$ **Swap** $\{x = y_0 \wedge y = x_0\}$.

Solution

$\{x = x_0 \wedge y = y_0\}$

$\{y = y_0 \wedge x = x_0\}$ Implied

$t := x;$

$\{y = y_0 \wedge t = x_0\}$ Assignment

$x := y;$

$\{x = y_0 \wedge t = x_0\}$ Assignment

$y := t;$

$\{x = y_0 \wedge y = x_0\}$ Assignment

- x_0 and y_0 are logical variables: they remember the initial values (recall **Fac**$_2$)!

Exercise 21.1: Swap without a temporary

Show the same triple for the program $x := x + y;\ y := x - y;\ x := x - y$.

Topic 21.1

Proof Outlines and Loop Invariants

Recall: the If-then-Else rule and its variant

The rule

$$\frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

Recall: the If-then-Else rule and its variant

The rule

$$\frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

The variant

$$\frac{\{\varphi_1\}\ C_1\ \{\psi\} \qquad \{\varphi_2\}\ C_2\ \{\psi\}}{\{(B \to \varphi_1) \wedge (\neg B \to \varphi_2)\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else (variant)}$$

Recall: the If-then-Else rule and its variant

The rule

$$\frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

The variant

$$\frac{\{\varphi_1\}\ C_1\ \{\psi\} \qquad \{\varphi_2\}\ C_2\ \{\psi\}}{\{(B \to \varphi_1) \wedge (\neg B \to \varphi_2)\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else (variant)}$$

- **Quick question:** we know ψ and the two branches, but not φ. Which rule helps us *compute* a precondition?

Recall: the If-then-Else rule and its variant

The rule

$$\frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

The variant

$$\frac{\{\varphi_1\}\ C_1\ \{\psi\} \qquad \{\varphi_2\}\ C_2\ \{\psi\}}{\{(B \to \varphi_1) \wedge (\neg B \to \varphi_2)\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else (variant)}$$

- **Quick question:** we know ψ and the two branches, but not φ. Which rule helps us *compute* a precondition?
- The variant: push ψ through C_1 and C_2 to get φ_1 and φ_2, then combine them as above.

Recall: the If-then-Else rule and its variant

The rule

$$\frac{\{\varphi \wedge B\}\ C_1\ \{\psi\} \qquad \{\varphi \wedge \neg B\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else}$$

The variant

$$\frac{\{\varphi_1\}\ C_1\ \{\psi\} \qquad \{\varphi_2\}\ C_2\ \{\psi\}}{\{(B \to \varphi_1) \wedge (\neg B \to \varphi_2)\}\ \textbf{if}\ B\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}}\ \text{If-then-Else (variant)}$$

- **Quick question:** we know ψ and the two branches, but not φ. Which rule helps us *compute* a precondition?
- The variant: push ψ through C_1 and C_2 to get φ_1 and φ_2, then combine them as above.
- Together with Implied, the two rules are equivalent (**Exercise 20.5**).

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Unk**$_1$ $\{y = x + 1\}$ holds

$a := x + 1;$

if $(a - 1 = 0)\{$

$y := 1;$

$\}$ **else** $\{$

$y := a;$

$\}$

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$y := 1;$

$\}$ **else** $\{$

$y := a;$

$\}$

$\{y = x + 1\}$

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$y := 1;$

$\}$ **else** $\{$

$y := a;$

$\}$

$\{y = x + 1\}$

If-then-Else

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$y := 1;$

$\}$ **else** $\{$

$y := a;$

$\}$

$\{y = x + 1\}$

- What is the weakest φ such that
  $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$
  holds?

If-then-Else

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$y := 1;$

$\{y = x + 1\}$

$\}$ **else** $\{$

$y := a;$

$\}$

$\{y = x + 1\}$

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1

If-then-Else

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$\{1 = x + 1\}$

$y := 1;$

$\{y = x + 1\}$ Assignment

$\}$ **else** $\{$

$y := a;$

$\}$

$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$\{1 = x + 1\}$

$y := 1;$

$\{y = x + 1\}$ Assignment

$\}$ **else** $\{$

$y := a;$

$\{y = x + 1\}$

$\}$

$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$\{1 = x + 1\}$

$y := 1;$

$\{y = x + 1\}$ Assignment

$\}$ **else** $\{$

$\{a = x + 1\}$

$y := a;$

$\{y = x + 1\}$ Assignment

$\}$

$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

if $(a - 1 = 0)\{$

$\{1 = x + 1\}$

$y := 1;$

$\{y = x + 1\}$ Assignment

$\}$ **else** $\{$

$\{a = x + 1\}$

$y := a;$

$\{y = x + 1\}$ Assignment

$\}$

$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2
- Set φ to be $(B \to \varphi_1) \wedge (\neg B \to \varphi_2)$ – the variant of the rule from Lecture 20

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

$\{((a - 1 = 0) \rightarrow (1 = x + 1)) \wedge (\neg(a - 1 = 0) \rightarrow (a = x + 1))\}$

if $(a - 1 = 0)\{$

$\{1 = x + 1\}$ If-then-Else

$y := 1;$

$\{y = x + 1\}$ Assignment

$\}$ **else** $\{$

$\{a = x + 1\}$ If-then-Else

$y := a;$

$\{y = x + 1\}$ Assignment

$\}$

$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2
- Set φ to be $(B \rightarrow \varphi_1) \wedge (\neg B \rightarrow \varphi_2)$ – the variant of the rule from Lecture 20

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$a := x + 1;$

$\{((a - 1 = 0) \to (1 = x + 1)) \wedge (\neg(a - 1 = 0) \to (a = x + 1))\}$ Assignment

if $(a - 1 = 0)\{$

$\{1 = x + 1\}$ If-then-Else

$y := 1;$

$\{y = x + 1\}$ Assignment

$\}$ **else** $\{$

$\{a = x + 1\}$ If-then-Else

$y := a;$

$\{y = x + 1\}$ Assignment

$\}$

$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2
- Set φ to be $(B \to \varphi_1) \wedge (\neg B \to \varphi_2)$ – the variant of the rule from Lecture 20

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$

$\{((x + 1 - 1 = 0) \to (1 = x + 1)) \wedge$
$(\neg(x + 1 - 1 = 0) \to (x + 1 = x + 1))\}$
$a := x + 1;$
$\{((a - 1 = 0) \to (1 = x + 1)) \wedge (\neg(a - 1 = 0) \to (a = x + 1))\}$ Assignment
if $(a - 1 = 0)\{$
$\{1 = x + 1\}$ If-then-Else
$y := 1;$
$\{y = x + 1\}$ Assignment
} **else** {
$\{a = x + 1\}$ If-then-Else
$y := a;$
$\{y = x + 1\}$ Assignment
}
$\{y = x + 1\}$ If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2
- Set φ to be $(B \to \varphi_1) \wedge (\neg B \to \varphi_2)$ – the variant of the rule from Lecture 20

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$	
$\{((x = 0) \to (0 = x)) \wedge (\neg(x = 0) \to (x + 1 = x + 1))\}$	Implied
$\{((x + 1 - 1 = 0) \to (1 = x + 1)) \wedge$ $(\neg(x + 1 - 1 = 0) \to (x + 1 = x + 1))\}$	Implied
$a := x + 1;$	
$\{((a - 1 = 0) \to (1 = x + 1)) \wedge (\neg(a - 1 = 0) \to (a = x + 1))\}$	Assignment
if $(a - 1 = 0)\{$	
$\{1 = x + 1\}$	If-then-Else
$y := 1;$	
$\{y = x + 1\}$	Assignment
$\}$ **else** $\{$	
$\{a = x + 1\}$	If-then-Else
$y := a;$	
$\{y = x + 1\}$	Assignment
$\}$	
$\{y = x + 1\}$	If-then-Else

- What is the weakest φ such that $\{\varphi\}$ **if** B $\{C_1\}$ **else** $\{C_2\}$ $\{\psi\}$ holds?
- Push ψ through C_1 to get φ_1
- Push ψ through C_2 to get φ_2
- Set φ to be $(B \to \varphi_1) \wedge (\neg B \to \varphi_2)$ – the variant of the rule from Lecture 20

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$	
$\{((x = 0) \rightarrow (0 = x)) \wedge (\neg(x = 0) \rightarrow (x + 1 = x + 1))\}$	Implied
$\{((x + 1 - 1 = 0) \rightarrow (1 = x + 1)) \wedge$	
$(\neg(x + 1 - 1 = 0) \rightarrow (x + 1 = x + 1))\}$	Implied
$a := x + 1;$	
$\{((a - 1 = 0) \rightarrow (1 = x + 1)) \wedge (\neg(a - 1 = 0) \rightarrow (a = x + 1))\}$	Assignment
if $(a - 1 = 0)\{$	
$\{1 = x + 1\}$	If-then-Else
$y := 1;$	
$\{y = x + 1\}$	Assignment
$\}$ **else** $\{$	
$\{a = x + 1\}$	If-then-Else
$y := a;$	
$\{y = x + 1\}$	Assignment
$\}$	
$\{y = x + 1\}$	If-then-Else

- **Quick question:** is $(B \rightarrow \varphi_1) \wedge (\neg B \rightarrow \varphi_2)$ the same as $(B \wedge \varphi_1) \vee (\neg B \wedge \varphi_2)$?

Using the If-then-Else rule. E.g., $\vdash_{par} \{\top\}$ **Succ** $\{y = x + 1\}$ holds

$\{\top\}$	
$\{((x = 0) \to (0 = x)) \wedge (\neg(x = 0) \to (x + 1 = x + 1))\}$	Implied
$\{((x + 1 - 1 = 0) \to (1 = x + 1)) \wedge$	
$(\neg(x + 1 - 1 = 0) \to (x + 1 = x + 1))\}$	Implied
$a := x + 1;$	
$\{((a - 1 = 0) \to (1 = x + 1)) \wedge (\neg(a - 1 = 0) \to (a = x + 1))\}$	Assignment
if $(a - 1 = 0)\{$	
$\{1 = x + 1\}$	If-then-Else
$y := 1;$	
$\{y = x + 1\}$	Assignment
$\}$ **else** $\{$	
$\{a = x + 1\}$	If-then-Else
$y := a;$	
$\{y = x + 1\}$	Assignment
$\}$	
$\{y = x + 1\}$	If-then-Else

- **Quick question:** is $(B \to \varphi_1) \wedge (\neg B \to \varphi_2)$ the same as $(B \wedge \varphi_1) \vee (\neg B \wedge \varphi_2)$?
- Pushing ψ backwards like this is Dijkstra's weakest precondition calculus (1975): φ is $wp(\textbf{if } \ldots, \psi)$.



Edsger W. Dijkstra (1930–2002)

Are these still relevant 50 years later?



DOI:10.1145/3777544

Formal verification of AI workflows.

BY ERIK MEIJER

Guardians of the Agents

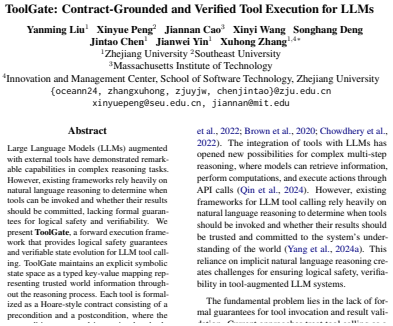
AGENTIC APPLICATIONS—AI SYSTEMS empowered to take autonomous actions by calling external tools—are the current rage in software development. They promise efficiency, convenience, and reduced human intervention. Giving autonomous agents access to

that these risks of ceding control to autonomous agents are sufficiently threatening that their use should be entirely prohibited.[a]

To mitigate the risks inherent in agentic applications, we propose a safety paradigm rooted in mathematical proof verification. In this design pattern, the AI agent is required to generate formal proofs demonstrating the safety of planned actions before being authorized to execute them. This approach parallels existing real-world practices (for example, credit card checksums, banknote security features), where complexity lies in production, but verification remains simple and efficient. Moreover, it extends the mechanism used by widely adopted computing platforms such as Java and .NET, where code undergoes a process of bytecode verification prior to execution.[b] Bytecode verification ensures crucial safety guarantees—notably memory safety, type correctness, and proper access control. Our proposal naturally extends this established principle, adapting it effectively to the emerging domain of agentic computations, thus providing robust assurances against harmful or unintended agent behavior.

The current state of the art in AI safety is to rely on evaluations (evals), which aim to identify potential risks, vulnerabilities, or harmful behaviors

E. Meijer, *Guardians of the Agents*, CACM 69(1), Jan. 2026



ToolGate: Contract-Grounded and Verified Tool Execution for LLMs

Yanming Liu[1] **Xinyue Peng**[2] **Jiannan Cao**[3] **Xinyi Wang**[4] **Songhang Deng**
Jintao Chen[1] **Jianwei Yin**[1] **Xuhong Zhang**[1,4*]
[1]Zhejiang University [2]Southeast University
[3]Massachusetts Institute of Technology
[4]Innovation and Management Center, School of Software Technology, Zhejiang University
{oceann24, zhangxuhong, zjuyjw, chenjintao}@zju.edu.cn
xinyuepeng@seu.edu.cn, jiannan@mit.edu

Abstract

Large Language Models (LLMs) augmented with external tools have demonstrated remarkable capabilities in complex reasoning tasks. However, existing frameworks rely heavily on natural language reasoning to determine when tools can be invoked and whether their results should be committed, lacking formal guarantees for logical safety and verifiability. We present **ToolGate**, a forward execution framework that provides logical safety guarantees and verifiable state evolution for LLM tool calling. ToolGate maintains an explicit symbolic state space as a typed key-value mapping representing trusted world information throughout the reasoning process. Each tool is formalized as a Hoare-style contract consisting of a precondition and a postcondition, where the

et al., 2022; Brown et al., 2020; Chowdhery et al., 2022). The integration of tools with LLMs has opened new possibilities for complex multi-step reasoning, where models can retrieve information, perform computations, and execute actions through API calls (Qin et al., 2024). However, existing frameworks for LLM tool calling rely heavily on natural language reasoning to determine when tools should be invoked and whether their results should be trusted and committed to the system's understanding of the world (Yang et al., 2024a). This reliance on implicit natural language reasoning creates challenges for ensuring logical safety, verifiability in tool-augmented LLM systems.

The fundamental problem lies in the lack of formal guarantees for tool invocation and result vali-

Liu et al., *ToolGate*, Findings of ACL 2026

Are these still relevant 50 years later?



DOI:10.1145/3777544

Formal verification of AI workflows.

BY ERIK MEIJER

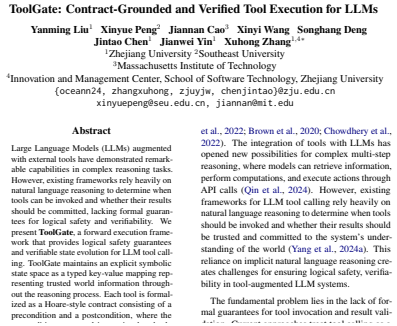
Guardians of the Agents

AGENTIC APPLICATIONS—AI SYSTEMS empowered to take autonomous actions by calling external tools—are the current rage in software development. They promise efficiency, convenience, and reduced human intervention. Giving autonomous agents access to that these risks of ceding control to autonomous agents are sufficiently threatening that their use should be entirely prohibited.[a]

To mitigate the risks inherent in agentic applications, we propose a safety paradigm rooted in mathematical proof verification. In this design pattern, the AI agent is required to generate formal proofs demonstrating the safety of planned actions before being authorized to execute them. This approach parallels existing real-world practices (for example, credit card checksums, banknote security features), where complexity lies in production, but verification remains simple and efficient. Moreover, it extends the mechanism used by widely adopted computing platforms such as Java and .NET, where code undergoes a process of bytecode verification prior to execution.[b] Bytecode verification ensures crucial safety guarantees—notably memory safety, type correctness, and proper access control. Our proposal naturally extends this established principle, adapting it effectively to the emerging domain of agentic computations, thus providing robust assurances against harmful or unintended agent behavior.

The current state of the art in AI safety is to rely on evaluations (evals), which aim to identify potential risks, vulnerabilities, or harmful behaviors

E. Meijer, *Guardians of the Agents*, CACM 69(1), Jan. 2026



ToolGate: Contract-Grounded and Verified Tool Execution for LLMs

Yanming Liu[1] Xinyue Peng[2] Jiannan Cao[3] Xinyi Wang Songhang Deng
Jintao Chen[1] Jianwei Yin[1] Xuhong Zhang[1,4*]
[1]Zhejiang University [2]Southeast University
[3]Massachusetts Institute of Technology
[4]Innovation and Management Center, School of Software Technology, Zhejiang University
{oceann24, zhangxuhong, zjuyjw, chenjintao}@zju.edu.cn
xinyuepeng@seu.edu.cn, jiannan@mit.edu

Abstract

Large Language Models (LLMs) augmented with external tools have demonstrated remarkable capabilities in complex reasoning tasks. However, existing frameworks rely heavily on natural language reasoning to determine when tools can be invoked and whether their results should be committed, lacking formal guarantees for logical safety and verifiability. We present **ToolGate**, a forward execution framework that provides logical safety guarantees and verifiable state evolution for LLM tool calling. ToolGate maintains an explicit symbolic state space as a typed key-value mapping representing trusted world information throughout the reasoning process. Each tool is formalized as a Hoare-style contract consisting of a precondition and a postcondition, where the

et al., 2022; Brown et al., 2020; Chowdhery et al., 2022). The integration of tools with LLMs has opened new possibilities for complex multi-step reasoning, where models can retrieve information, perform computations, and execute actions through API calls (Qin et al., 2024). However, existing frameworks for LLM tool calling rely heavily on natural language reasoning to determine when tools should be invoked and whether their results should be trusted and committed to the system's understanding of the world (Yang et al., 2024a). This reliance on implicit natural language reasoning creates challenges for ensuring logical safety, verifiability in tool-augmented LLM systems.

The fundamental problem lies in the lack of formal guarantees for tool invocation and result vali-

Liu et al., *ToolGate*, Findings of ACL 2026

verification process relies on preconditions, postconditions, and invariants, ensuring only mathematically safe workflows proceed to execution.

CACM 2026, p. 51

Tackling the partial while!

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

Tackling the partial while!

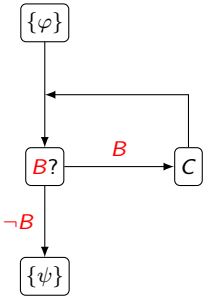
$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

- The rule helps if we start with $\{\eta\}$ **while** B $\{C\}$ $\{\eta \wedge \neg B\}$.

Tackling the partial while!

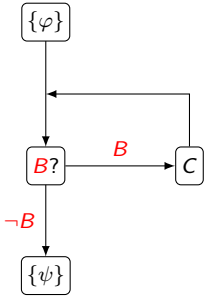
$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

- The rule helps if we start with $\{\eta\}$ **while** B $\{C\}$ $\{\eta \wedge \neg B\}$.
- But we have just $\{\varphi\}$ **while** B $\{C\}$ $\{\psi\}$, where φ and ψ are not directly related?!



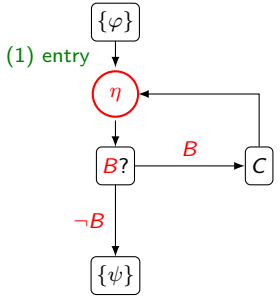
Tackling the partial while!

- **Quick recall:** which rule links φ and ψ to the loop rule?



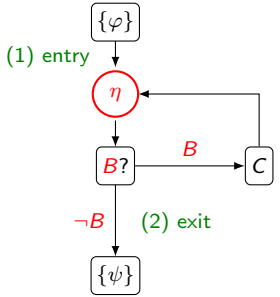
Tackling the partial while!

- **Quick recall:** which rule links φ and ψ to the loop rule?
- Implied! So find η such that:
  1. $\vdash_{AR} \varphi \to \eta$



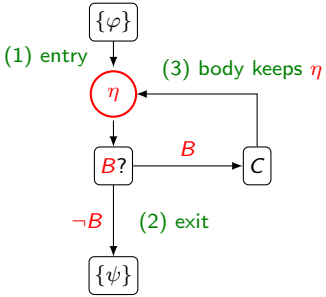
Tackling the partial while!

- **Quick recall:** which rule links φ and ψ to the loop rule?
- Implied! So find η such that:
  1. $\vdash_{AR} \varphi \to \eta$
  2. $\vdash_{AR} \eta \wedge \neg B \to \psi$



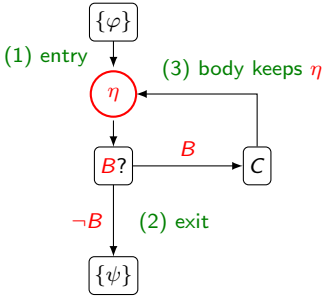
Tackling the partial while!

- **Quick recall:** which rule links φ and ψ to the loop rule?
- Implied! So find η such that:
  1. $\vdash_{AR} \varphi \to \eta$
  2. $\vdash_{AR} \eta \wedge \neg B \to \psi$
  3. $\vdash_{par} \{\eta \wedge B\}\ C\ \{\eta\}$



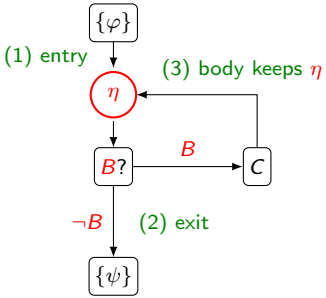
Tackling the partial while!

- **Quick recall:** which rule links φ and ψ to the loop rule?
- Implied! So find η such that:
  1. $\vdash_{AR} \varphi \to \eta$
  2. $\vdash_{AR} \eta \wedge \neg B \to \psi$
  3. $\vdash_{par} \{\eta \wedge B\}\ C\ \{\eta\}$
- Then Partial-while and Implied give $\vdash_{par} \{\varphi\}$ **while** $B\ \{C\}\ \{\psi\}$. Finding η needs ingenuity, unlike the other cases!



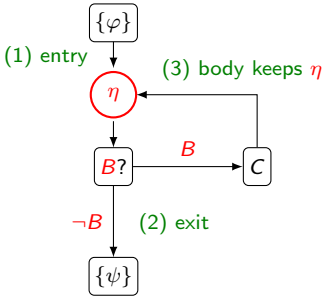
Tackling the partial while!

- An invariant η holds before and after every iteration, even if not during! It is an induction hypothesis on the number of iterations: (1) is the base case, (3) the inductive step.



Tackling the partial while!

- An invariant η holds before and after every iteration, even if not during! It is an induction hypothesis on the number of iterations: (1) is the base case, (3) the inductive step.
- **Exercise 21.2.** Give an example of a trivial invariant for any loop! Why does that not suffice?



The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop
iterations	0
x	6
y	1
z	0

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop	
iterations	0	1
x	6	6
y	1	1
z	0	1

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop		
iterations	0	1	2
x	6	6	6
y	1	1	2
z	0	1	2

The factorial example revisited

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop			
iterations	0	1	2	3
x	6	6	6	6
y	1	1	2	6
z	0	1	2	3

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop				
iterations	0	1	2	3	4
x	6	6	6	6	6
y	1	1	2	6	24
z	0	1	2	3	4

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop					
iterations	0	1	2	3	4	5
x	6	6	6	6	6	6
y	1	1	2	6	24	120
z	0	1	2	3	4	5

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.

	before loop						*after loop*
iterations	0	1	2	3	4	5	6
x	6	6	6	6	6	6	6
y	1	1	2	6	24	120	720
z	0	1	2	3	4	5	6

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.
- Notice that $y = z!$ holds every time the guard is tested!

	before loop						*after loop*
iterations	0	1	2	3	4	5	6
x	6	6	6	6	6	6	6
y	1	1	2	6	24	120	720
z	0	1	2	3	4	5	6
$z!$	1	1	2	6	24	120	720

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.
- Notice that $y = z!$ holds every time the guard is tested!

Quick question: does $y = z!$ hold *between* $z := z + 1$ and $y := y * z$?

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.
- Notice that $y = z!$ holds every time the guard is tested!

Quick question: does $y = z!$ hold *between* $z := z + 1$ and $y := y * z$? No! There $y = (z - 1)!$: the invariant may be broken *during* the body.

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.
- Notice that $y = z!$ holds every time the guard is tested!
- But for this to be our loop inv candidate we should check 2 other things.

The factorial example revisited

Fac$_1$	$y := 1;$
	$z := 0;$
inp x	**while** $(z \neq x)\{$
out y	$\quad z := z + 1;$
	$\quad y := y * z;$
	$\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.
- Notice that $y = z!$ holds every time the guard is tested!
- But for this to be our loop inv candidate we should check 2 other things.

- (1) Pre-condn of while loop implies it: $\vdash_{AR} (y = 1 \wedge z = 0) \rightarrow (y = z!)$ is valid.
- (2) Together with neg of guard, implies post-condn: $\vdash_{AR} (y = z! \wedge x = z) \rightarrow (y = x!)$ is valid.

The factorial example revisited

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the invariant here?
- **Exercise 21.3**
  - Starting with $x = 6$, write the values of x, y, z before the loop and after each iteration.
- Notice that $y = z!$ holds every time the guard is tested!
- But for this to be our loop inv candidate we should check 2 other things.

- (1) Pre-condn of while loop implies it: $\vdash_{AR} (y = 1 \wedge z = 0) \rightarrow (y = z!)$ is valid.
- (2) Together with neg of guard, implies post-condn: $\vdash_{AR} (y = z! \wedge x = z) \rightarrow (y = x!)$ is valid.

So, let's write the full proof for the factorial program now!

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\quad z := z + 1$

$\quad y := y * z$

$\}$

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\quad z := z + 1$

$\quad y := y * z$

$\{y = z!\}$

$\}$

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\quad z := z + 1$

$\{y \cdot z = z!\}$

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$y := 1;$

$z := 0;$

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

$y := 1;$

$z := 0;$
$\{y = z!\}$
while $(z \neq x)\{$
$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard
$\{y \cdot (z+1) = (z+1)!\}$ Implied
$\quad z := z + 1$
$\{y \cdot z = z!\}$ Assignment
$\quad y := y * z$
$\{y = z!\}$ Assignment
$\}$
$\{y = z! \wedge \neg(z \neq x)\}$ Partial-while

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

$\{\top\}$

$y := 1;$

$z := 0;$

$\{y = z!\}$

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \wedge \neg(z \neq x)\}$ Partial-while

$\{y = x!\}$

The factorial example solved!

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

$\{\top\}$

$y := 1;$

$z := 0;$

$\{y = z!\}$

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \wedge \neg(z \neq x)\}$ Partial-while

$\{y = x!\}$ Implied

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

$\{\top\}$

$y := 1;$

$\{y = 0!\}$

$z := 0;$

$\{y = z!\}$ Assignment

while $(z \neq x)\{$

$\{y = z! \wedge z \neq x\}$ Inv Hyp. and Guard

$\{y \cdot (z + 1) = (z + 1)!\}$ Implied

$\quad z := z + 1$

$\{y \cdot z = z!\}$ Assignment

$\quad y := y * z$

$\{y = z!\}$ Assignment

$\}$

$\{y = z! \wedge \neg(z \neq x)\}$ Partial-while

$\{y = x!\}$ Implied

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

$\{\top\}$	
$\{1 = 0!\}$	
$y := 1;$	
$\{y = 0!\}$	Assignment
$z := 0;$	
$\{y = z!\}$	Assignment
while $(z \neq x)\{$	
$\{y = z! \wedge z \neq x\}$	Inv Hyp. and Guard
$\{y \cdot (z + 1) = (z + 1)!\}$	Implied
$\quad z := z + 1$	
$\{y \cdot z = z!\}$	Assignment
$\quad y := y * z$	
$\{y = z!\}$	Assignment
$\}$	
$\{y = z! \wedge \neg(z \neq x)\}$	Partial-while
$\{y = x!\}$	Implied

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

$\{\top\}$	
$\{1 = 0!\}$	Implied
$y := 1;$	
$\{y = 0!\}$	Assignment
$z := 0;$	
$\{y = z!\}$	Assignment
while $(z \neq x)\{$	
$\{y = z! \wedge z \neq x\}$	Inv Hyp. and Guard
$\{y \cdot (z + 1) = (z + 1)!\}$	Implied
$\quad z := z + 1$	
$\{y \cdot z = z!\}$	Assignment
$\quad y := y * z$	
$\{y = z!\}$	Assignment
$\}$	
$\{y = z! \wedge \neg(z \neq x)\}$	Partial-while
$\{y = x!\}$	Implied

The factorial example solved!

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}$$

Partial-while applied, with $\eta := y = z!$ and $B := z \neq x$

(1) entry, (2) exit, (3) body keeps η

	$\{\top\}$	
	$\{1 = 0!\}$	Implied
	$y := 1;$	
	$\{y = 0!\}$	Assignment
	$z := 0;$	
(1)	$\{y = z!\}$	Assignment
	while $(z \neq x)\{$	
(3)	$\{y = z! \wedge z \neq x\}$	Inv Hyp. and Guard
	$\{y \cdot (z + 1) = (z + 1)!\}$	Implied
	$\quad z := z + 1$	
	$\{y \cdot z = z!\}$	Assignment
	$\quad y := y * z$	
	$\{y = z!\}$	Assignment
	$\}$	
(2)	$\{y = z! \wedge \neg(z \neq x)\}$	Partial-while
	$\{y = x!\}$	Implied

Another Factorial Example

Fac$_2$ *inp* x *out* y	$y := 1$; **while** $(x \neq 0)\{$ $\quad y := y * x$; $\quad x := x - 1$; $\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

Another Factorial Example

Fac$_2$	$y := 1;$
inp x	**while** $(x \neq 0)\{$
out y	$\quad y := y * x;$
	$\quad x := x - 1;$
	$\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

In practice, use SMT Solvers to automate the checks.

- Proof steps like these can be handed to an SMT solver such as Z3 (a wrong invariant, e.g. $y = x!$, then gets a counterexample).

Another Factorial Example

Fac$_2$	$y := 1;$
inp x	**while** $(x \neq 0)\{$
out y	$\quad y := y * x;$
	$\quad x := x - 1;$
	$\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

In practice, use SMT Solvers to automate the checks.

- Proof steps like these can be handed to an SMT solver such as Z3 (a wrong invariant, e.g. $y = x!$, then gets a counterexample). Which theory is needed?

Another Factorial Example

Fac$_2$	$y := 1;$
inp x	**while** $(x \neq 0)\{$
out y	$\quad y := y * x;$
	$\quad x := x - 1;$
	$\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

In practice, use SMT Solvers to automate the checks.

- Proof steps like these can be handed to an SMT solver such as Z3 (a wrong invariant, e.g. $y = x!$, then gets a counterexample). Which theory is needed?
- Can it *synthesize* invariants?

Another Factorial Example

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

In practice, use SMT Solvers to automate the checks.

- Proof steps like these can be handed to an SMT solver such as Z3 (a wrong invariant, e.g. $y = x!$, then gets a counterexample). Which theory is needed?
- Can it *synthesize* invariants? Not always!

Another Factorial Example

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

In practice, use SMT Solvers to automate the checks.

- Proof steps like these can be handed to an SMT solver such as Z3 (a wrong invariant, e.g. $y = x!$, then gets a counterexample). Which theory is needed?
- Can it *synthesize* invariants? Not always! You (or an LLM?) have to provide it.

Another Factorial Example

Fac$_2$ *inp* x *out* y	$y := 1;$ **while** $(x \neq 0)\{$ $\quad y := y * x;$ $\quad x := x - 1;$ $\}$

Exercise 21.4 Consider program **Fac**$_2$

1. Does the same invariant (as for **Fac**$_1$) work for this? Why or why not?
2. Find a correct invariant (hint: use x_0) and show that
   $\vdash_{par} \{x = x_0 \wedge x \geq 0\}$ **Fac**$_2$ $\{y = x_0!\}$

In practice, use SMT Solvers to automate the checks.

- Proof steps like these can be handed to an SMT solver such as Z3 (a wrong invariant, e.g. $y = x!$, then gets a counterexample). Which theory is needed?
- Can it *synthesize* invariants? Not always! You (or an LLM?) have to provide it.
- Optional demo exercise with Z3 in next problem sheet!

Topic 21.2

Total Correctness

Total correctness

- **Quick question:** what happens if we run $\mathbf{Fac}_1$ with $x = -1$?

Total correctness

- **Quick question:** what happens if we run $\mathbf{Fac}_1$ with $x = -1$?
- The invariant $y = z!$ keeps holding, but z never reaches x: the loop *never exits*! So $\{\top\}\ \mathbf{Fac}_1\ \{y = x!\}$ is only partially correct.

Total correctness

- **Quick question:** what happens if we run $\mathbf{Fac}_1$ with $x = -1$?
- The invariant $y = z!$ keeps holding, but z never reaches x: the loop *never exits*! So $\{\top\}\ \mathbf{Fac}_1\ \{y = x!\}$ is only partially correct.
- The partial correctness rules developed till now do not say anything if a program P loops indefinitely.

Total correctness

- **Quick question:** what happens if we run $\mathbf{Fac}_1$ with $x = -1$?
- The invariant $y = z!$ keeps holding, but z never reaches x: the loop *never exits*! So $\{\top\}\ \mathbf{Fac}_1\ \{y = x!\}$ is only partially correct.
- The partial correctness rules developed till now do not say anything if a program P loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?

Total correctness

- **Quick question:** what happens if we run $\mathbf{Fac}_1$ with $x = -1$?
- The invariant $y = z!$ keeps holding, but z never reaches x: the loop *never exits*! So $\{\top\}\ \mathbf{Fac}_1\ \{y = x!\}$ is only partially correct.
- The partial correctness rules developed till now do not say anything if a program P loops indefinitely.
- Can we extend the proof calculus to show that programs also terminate?
- The only rule that causes looping is the while condn, so the only thing to change is the while rule!

Total correctness: the core idea

Core idea!

- for partial correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.

Total correctness: the core idea

Core idea!

- for partial correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.
  - In fact, we look for an integer expression that decreases every loop iteration, but always remains non-negative.

Total correctness: the core idea

Core idea!

- for partial correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.
  - In fact, we look for an integer expression that decreases every loop iteration, but always remains non-negative.
  - hence, it can decrease only finitely many times and hence the loop terminates.

Total correctness: the core idea

Core idea!

- for partial correctness of while loop, we have an invariant.
- for termination of while loop, we need something that varies but only a bounded no. of times.
  - In fact, we look for an integer expression that decreases every loop iteration, but always remains non-negative.
  - hence, it can decrease only finitely many times and hence the loop terminates.
  - Such an expression is called a variant.



Revisiting the Factorial example

Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the variant here?
- Can you find an integer expression whose value decreases at each loop iteration?

Revisiting the Factorial example

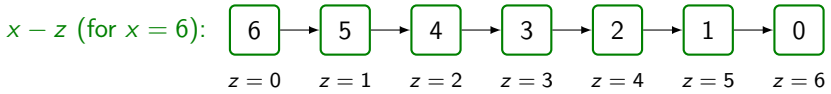
Fac$_1$ *inp* x *out* y	$y := 1;$ $z := 0;$ **while** $(z \neq x)\{$ $\quad z := z + 1;$ $\quad y := y * z;$ $\}$

- What is the variant here?
- Can you find an integer expression whose value decreases at each loop iteration?
- easy! the expression $x - z$.

Revisiting the Factorial example

	$y := 1;$
	$z := 0;$
Fac$_1$	**while** $(z \neq x)\{$
inp x	$\quad z := z + 1;$
out y	$\quad y := y * z;$
	$\}$

- What is the variant here?
- Can you find an integer expression whose value decreases at each loop iteration?
- easy! the expression $x - z$.
- Note that it is always nonnegative (as long as $x \geq 0$ initially).
- When it is 0, the loop terminates!



Proof rule for total correctness of while loops

For while loops

Total correctness = Partial correctness + Variant

Proof rule for total correctness of while loops

For while loops

Total correctness = Partial correctness + Variant

$$\frac{\{\eta \wedge B\}\ C\ \{\eta\}}{\{\eta\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Partial-while}$$

Proof rule for total correctness of while loops

For while loops

Total correctness = Partial correctness + Variant

$$\frac{\{\eta \wedge B \wedge 0 \leq E = E_0\}\ C\ \{\eta \wedge 0 \leq E < E_0\}}{\{\eta \wedge 0 \leq E\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Total-while}$$

Proof rule for total correctness of while loops

For while loops

Total correctness = Partial correctness + Variant

$$\frac{\{\eta \wedge B \wedge 0 \leq E = E_0\}\ C\ \{\eta \wedge 0 \leq E < E_0\}}{\{\eta \wedge 0 \leq E\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Total-while}$$

- Variant E is the integer expression
  - whose value is equal to the logical variable E_0 before the loop body is executed
  - and after one execution of the loop body, the value is strictly less than E_0 but still non-negative.

Proof rule for total correctness of while loops

For while loops

Total correctness = Partial correctness + Variant

$$\frac{\{\eta \wedge B \wedge 0 \leq E = E_0\}\ C\ \{\eta \wedge 0 \leq E < E_0\}}{\{\eta \wedge 0 \leq E\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Total-while}$$

- Variant E is the integer expression
  - whose value is equal to the logical variable E_0 before the loop body is executed
  - and after one execution of the loop body, the value is strictly less than E_0 but still non-negative.
- η is the invariant as before

Proof rule for total correctness of while loops

For while loops

Total correctness = Partial correctness + Variant

$$\frac{\{\eta \wedge B \wedge 0 \leq E = E_0\}\ C\ \{\eta \wedge 0 \leq E < E_0\}}{\{\eta \wedge 0 \leq E\}\ \textbf{while}\ B\ \{C\}\ \{\eta \wedge \neg B\}}\ \text{Total-while}$$

- Variant E is the integer expression
  - whose value is equal to the logical variable E_0 before the loop body is executed
  - and after one execution of the loop body, the value is strictly less than E_0 but still non-negative.
- η is the invariant as before
- For $\mathbf{Fac}_1$: $E = x - z$