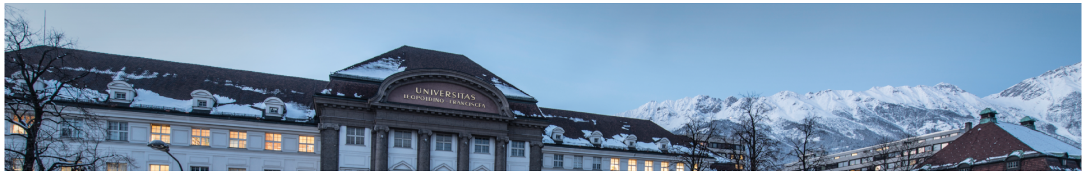
universität innsbruck

Summer Term 2024




# Program Verification

## Part 6 – Verification of Imperative Programs

René Thiemann

Department of Computer Science

# Imperative Programs



## Imperative Programs

- we here consider a small imperative programming language
- it consists of
  - arithmetic expressions $\mathcal{A}$ over some set of variables $\mathcal{V}$

$$\frac{n \in \mathbb{Z}}{n \in \mathcal{A}} \qquad \frac{x \in \mathcal{V}}{x \in \mathcal{A}} \qquad \frac{\{e_1, e_2\} \subseteq \mathcal{A} \quad \odot \in \{\texttt{+},\texttt{-},\texttt{*}\}}{e_1 \odot e_2 \in \mathcal{A}}$$

  - Boolean expressions $\mathcal{B}$

$$\frac{c \in \{\texttt{true}, \texttt{false}\}}{c \in \mathcal{B}} \qquad \frac{\{e_1, e_2\} \subseteq \mathcal{A} \quad \odot \in \{\texttt{=},\texttt{<},\texttt{<=},\texttt{!=}\}}{e_1 \odot e_2 \in \mathcal{B}}$$

$$\frac{b \in \mathcal{B}}{\texttt{!}\,b \in \mathcal{B}} \qquad \frac{\{b_1, b_2\} \subseteq \mathcal{B} \quad \odot \in \{\texttt{\&\&},\texttt{||}\}}{b_1 \odot b_2 \in \mathcal{B}}$$

  - commands $\mathcal{C}$





## Commands and Programs

- commands $\mathcal{C}$ consist of
  - assignments
$$\frac{x \in \mathcal{V} \quad e \in \mathcal{A}}{x := e \in \mathcal{C}}$$
  - if-then-else
$$\frac{b \in \mathcal{B} \quad \{C_1, C_2\} \subseteq \mathcal{C}}{\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \in \mathcal{C}}$$
  - sequential execution
$$\frac{\{C_1, C_2\} \subseteq \mathcal{C}}{C_1 \,\texttt{;}\, C_2 \in \mathcal{C}}$$
  - while-loops
$$\frac{b \in \mathcal{B} \quad C \in \mathcal{C}}{\texttt{while}\ b\ \{C\} \in \mathcal{C}}$$
  - no-operation
$$\frac{}{\texttt{skip} \in \mathcal{C}}$$
  - curly braces are added for disambiguation, e.g. consider `while x < 5 { x := x + 2 } ; y := y - 1`
- a program $P$ is just a command $C$

## Verification

- partial correctness predicate via Hoare-triples: $\models (\!|\varphi|\!)\ P\ (\!|\psi|\!)$
  - semantic notion
  - meaning: whenever initial state satisfies $\varphi$,
  - and execution of $P$ terminates,
  - then final state satisfies $\psi$
  - $\varphi$ is called precondition, $\psi$ is postcondition
  - here, formulas may range over program variables and logical variables
  - clearly, $\models$ requires semantic of commands
- Hoare calculus: $\vdash (\!|\varphi|\!)\ P\ (\!|\psi|\!)$
  - syntactic calculus (similar to natural deduction)
  - sound: whenever $\vdash (\!|\varphi|\!)\ P\ (\!|\psi|\!)$ then $\models (\!|\varphi|\!)\ P\ (\!|\psi|\!)$





## Semantics – Expressions

- state is evaluation $\alpha : \mathcal{V} \to \mathbb{Z}$
- semantics of arithmetic and Boolean expressions are defined as
  - $[\![\cdot]\!]_\alpha : \mathcal{A} \to \mathbb{Z}$
    e.g., if $\alpha(x) = 5$ then $[\![6 * x + 1]\!]_\alpha = 31$
  - $[\![\cdot]\!]_\alpha : \mathcal{B} \to \{\mathsf{true}, \mathsf{false}\}$
    e.g., if $\alpha(x) = 5$ then $[\![6 * x + 1 < 20]\!]_\alpha = \mathsf{false}$
- we omit the straight-forward recursive definitions of $[\![\cdot]\!]_\alpha$ here





## Semantics – Commands

- semantics of commands is given via small-step-semantics
  defined as relation $\hookrightarrow \subseteq (\mathcal{C} \times (\mathcal{V} \to \mathbb{Z}))^2$

$$\frac{}{(x := e, \alpha) \hookrightarrow (\mathtt{skip}, \alpha[x := [\![e]\!]_\alpha])}$$

$$\frac{[\![b]\!]_\alpha = \mathsf{true}}{(\mathtt{if}\ b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, \alpha) \hookrightarrow (C_1, \alpha)}$$

$$\frac{[\![b]\!]_\alpha = \mathsf{false}}{(\mathtt{if}\ b\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2, \alpha) \hookrightarrow (C_2, \alpha)}$$

$$\frac{(C_1, \alpha) \hookrightarrow (C_1', \beta)}{(C_1; C_2, \alpha) \hookrightarrow (C_1'; C_2, \beta)} \qquad \frac{}{(\mathtt{skip}; C, \alpha) \hookrightarrow (C, \alpha)}$$

$$\frac{[\![b]\!]_\alpha = \mathsf{true}}{(\mathtt{while}\ b\ C, \alpha) \hookrightarrow (C; \mathtt{while}\ b\ C, \alpha)}$$

$$\frac{[\![b]\!]_\alpha = \mathsf{false}}{(\mathtt{while}\ b\ C, \alpha) \hookrightarrow (\mathtt{skip}, \alpha)}$$

- $(\mathtt{skip}, \alpha)$ is normal form





## Semantics – Programs

- we can formally define $\models (\!|\varphi|\!)\ P\ (\!|\psi|\!)$ as

$$\forall \alpha, \beta.\ \alpha \models \varphi \longrightarrow (P, \alpha) \hookrightarrow^* (\mathtt{skip}, \beta) \longrightarrow \beta \models \psi$$

- example specification: $(\!|x > 0|\!)\ P\ (\!|y \cdot y < x|\!)$
  - if initially $x > 0$, after running the program $P$,
    the final values of $x$ and $y$ must satisfy $y \cdot y < x$
  - nothing is required if initially $x \leq 0$
  - nothing is required if program does not terminate
  - specification is satisfied by program $P$ defined as

```
y := 0
```

  - specification is satisfied by program $P$ defined as

```
y := 0;
while (y * y < x) {
  y := y + 1
};
y := y - 1
```

## Program Variables and Logical Variables

- consider program $\mathit{Fact}$

```
y := 1;
while (x != 0) {
  y := y * x;
  x := x - 1
}
```

- specification for factorial: does $\models \llparenthesis x \geq 0 \rrparenthesis \; \mathit{Fact} \; \llparenthesis y = x! \rrparenthesis$ hold?
  - if $\alpha(x) = 6$ and $(\mathit{Fact}, \alpha) \hookrightarrow^* (\texttt{skip}, \beta)$ then $\beta(y) = 720 = 6!$
  - problem: $\beta(x) = 0$, so $y = x!$ does not hold for final values
  - hence $\not\models \llparenthesis x \geq 0 \rrparenthesis \; \mathit{Fact} \; \llparenthesis y = x! \rrparenthesis$, since specification is wrong
- solution: store initial values in logical variables
- in example: introduce logical variable $x_0$

$$\models \llparenthesis x = x_0 \wedge x \geq 0 \rrparenthesis \; \mathit{Fact} \; \llparenthesis y = x_0! \rrparenthesis$$

via logical variables we can refer to initial values



## Hoare Calculus



## A Calculus for Program Verification

- aim: syntax directed calculus to reason about programs
- Hoare calculus separates reasoning on programs from logical reasoning (arithmetic, ...)
- present calculus as overview now, then explain single rules

$$\frac{\vdash \llparenthesis \varphi \rrparenthesis \; C_1 \; \llparenthesis \eta \rrparenthesis \quad \vdash \llparenthesis \eta \rrparenthesis \; C_2 \; \llparenthesis \psi \rrparenthesis}{\vdash \llparenthesis \varphi \rrparenthesis \; C_1; C_2 \; \llparenthesis \psi \rrparenthesis} \; \text{composition}$$

$$\frac{}{\vdash \llparenthesis \varphi[x/e] \rrparenthesis \; x := e \; \llparenthesis \varphi \rrparenthesis} \; \text{assignment}$$

$$\frac{\vdash \llparenthesis \varphi \wedge b \rrparenthesis \; C_1 \; \llparenthesis \psi \rrparenthesis \quad \vdash \llparenthesis \varphi \wedge \neg b \rrparenthesis \; C_2 \; \llparenthesis \psi \rrparenthesis}{\vdash \llparenthesis \varphi \rrparenthesis \; \texttt{if } b \texttt{ then } C_1 \texttt{ else } C_2 \; \llparenthesis \psi \rrparenthesis} \; \text{if-then-else}$$

$$\frac{\vdash \llparenthesis \varphi \wedge b \rrparenthesis \; C \; \llparenthesis \varphi \rrparenthesis}{\vdash \llparenthesis \varphi \rrparenthesis \; \texttt{while } b \; C \; \llparenthesis \varphi \wedge \neg b \rrparenthesis} \; \text{while}$$

$$\frac{\models \varphi \longrightarrow \varphi' \quad \vdash \llparenthesis \varphi' \rrparenthesis \; C \; \llparenthesis \psi' \rrparenthesis \quad \models \psi' \longrightarrow \psi}{\vdash \llparenthesis \varphi \rrparenthesis \; C \; \llparenthesis \psi \rrparenthesis} \; \text{implication}$$

- read rules bottom up: in order to get lower part, prove upper part





## Composition Rule

$$\frac{\vdash \llparenthesis \varphi \rrparenthesis \; C_1 \; \llparenthesis \eta \rrparenthesis \quad \vdash \llparenthesis \eta \rrparenthesis \; C_2 \; \llparenthesis \psi \rrparenthesis}{\vdash \llparenthesis \varphi \rrparenthesis \; C_1; C_2 \; \llparenthesis \psi \rrparenthesis} \; \text{composition}$$

- applicability: whenever command is sequential composition $C_1; C_2$
- precondition is $\varphi$ and aim is to show that $\psi$ holds after execution
- rationale: find some midcondition $\eta$ such that execution of $C_1$ guarantees $\eta$, which can then be used as precondition to conclude $\psi$ after execution of $C_2$
- automation: finding suitable $\eta$ is usually automatic, see later slides

## Assignment Rule

$$\frac{}{\vdash (\!|\varphi[x/e]|\!)\; x := e\; (\!|\varphi|\!)}\ \text{assignment}$$

- applicability: whenever command is an assignment $x := e$
- to prove $\varphi$ after execution, show $\varphi[x/e]$ before execution
- substitution seems to be on wrong side
  - effect of assignment is substitution $x/e$, so shouldn't rule be $\vdash (\!|\varphi|\!)\; x := e\; (\!|\varphi[x/e]|\!)$ ? No, this reversed rule would be wrong
    - assume before executing $x := 5$, the value of $x$ is 6
    - before execution $\varphi = (x = 6)$ is satisfied, but after execution $\varphi[x/e] = (5 = 6)$ is not satisfied
- correct argumentation works as follows
  - if we want to ensure $\varphi$ after the assignment then we need to ensure that the resulting situation ($\varphi[x/e]$) holds before
  - correct examples
    - $\vdash (\!|2 = 2|\!)\; x := 2\; (\!|x = 2|\!)$
    - $\vdash (\!|2 = 4|\!)\; x := 2\; (\!|x = 4|\!)$
    - $\vdash (\!|2 - y > 2^2|\!)\; x := 2\; (\!|x - y > x^2|\!)$
- applying rule is easy when read from right to left: just substitute





## If-Then-Else Rule

$$\frac{\vdash (\!|\varphi \wedge b|\!)\; C_1\; (\!|\psi|\!) \quad \vdash (\!|\varphi \wedge \neg b|\!)\; C_2\; (\!|\psi|\!)}{\vdash (\!|\varphi|\!)\; \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\; (\!|\psi|\!)}\ \text{if-then-else}$$

- applicability: whenever command is an if-then-else
- effect:
  - the preconditions in the two branches are strengthened by adding the corresponding (negated) condition $b$ of the if-then-else
  - often the addition of $b$ and $\neg b$ is crucial to be able to perform the proofs for the Hoare-triples of $C_1$ and $C_2$, respectively
- rationale: if $b$ is true in some state, then the execution will choose $C_1$ and we can add $b$ as additional assumption; similar for other case
- applying rule is trivial from right to left





## While Rule

$$\frac{\vdash (\!|\varphi \wedge b|\!)\; C\; (\!|\varphi|\!)}{\vdash (\!|\varphi|\!)\; \texttt{while}\ b\ C\; (\!|\varphi \wedge \neg b|\!)}\ \text{while}$$

- applicability: only rule that handles while-loop
- key ingredient: loop invariant $\varphi$
- rationale
  - $\varphi$ is precondition, so in particular satisfied before loop execution
  - $\vdash (\!|\varphi \wedge b|\!)\; C\; (\!|\varphi|\!)$ ensures, that when entering the loop, $\varphi$ will be satisfied after one execution of the loop body $C$
  - in total, $\varphi$ will be satisfied after each loop iteration
  - hence, when leaving the loop, $\varphi$ and $\neg b$ are satisfied
  - while-rule does not enforce termination, partial correctness!
- automation
  - not automatic, since usually $\varphi$ is not provided and postcondition is not of form $\varphi \wedge \neg b$; example: $\vdash (\!|x = x_0 \wedge x \geq 0|\!)\; \mathit{Fact}\; (\!|y = x_0!|\!)$
  - finding suitable $\varphi$ is hard and needs user guidance





## Implication Rule

$$\frac{\models \varphi \longrightarrow \varphi' \quad \vdash (\!|\varphi'|\!)\; C\; (\!|\psi'|\!) \quad \models \psi' \longrightarrow \psi}{\vdash (\!|\varphi|\!)\; C\; (\!|\psi|\!)}\ \text{implication}$$

- applicability: every command; does not change command
- rationale: weakening precondition or strengthening postcondition is sound
- remarks
  - only rule which does not decompose commands
  - application relies on prover for underlying logic, i.e., one which can prove implications
  - three main applications
    - simplify conditions that arise from applying other rules in order to get more readable proofs, e.g., replace $x + 1 = y - 2$ by $x = y - 3$
    - prepare invariants, e.g., change postcondition from $\psi$ to some formula $\psi'$ of form $\chi \wedge \neg b$
    - core reasoning engine when closing proofs for while-loops in proof tableaux, see later slides

## Example Proof

$$\dfrac{prf_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (\!|(y \cdot x) \cdot (x-1)! = x_0! \land x - 1 \geq 0|\!)\, \texttt{y := y * x}\, (\!|y \cdot (x-1)! = x_0! \land x - 1 \geq 0|\!)}}{\vdash (\!|y \cdot x! = x_0! \land x \geq 0 \land x \neq 0|\!)\, \texttt{y := y * x}\, (\!|y \cdot (x-1)! = x_0! \land x - 1 \geq 0|\!)} \quad prf_2}{\vdash (\!|y \cdot x! = x_0! \land x \geq 0 \land x \neq 0|\!)\, \texttt{y := y * x; x := x - 1}\, (\!|y \cdot x! = x_0! \land x \geq 0|\!)}}{\vdash (\!|y \cdot x! = x_0! \land x \geq 0|\!)\, \texttt{while x != 0 \{y := y * x; x := x - 1\}}\, (\!|y \cdot x! = x_0! \land x \geq 0 \land \neg x \neq 0|\!)}}{\vdash (\!|y \cdot x! = x_0! \land x \geq 0|\!)\, \texttt{while x != 0 \{y := y * x; x := x - 1\}}\, (\!|y = x_0!|\!)}}{\vdash (\!|x = x_0 \land x \geq 0|\!)\, \texttt{y := 1;while x != 0 \{y := y * x; x := x - 1\}}\, (\!|y = x_0!|\!)}$$

where $prf_1$ is the following proof

$$\dfrac{\dfrac{}{\vdash (\!|1 \cdot x! = x_0! \land x \geq 0|\!)\, \texttt{y := 1}\, (\!|y \cdot x! = x_0! \land x \geq 0|\!)}}{\vdash (\!|x = x_0 \land x \geq 0|\!)\, \texttt{y := 1}\, (\!|y \cdot x! = x_0! \land x \geq 0|\!)}$$

and $prf_2$ is the following proof

$$\dfrac{}{\vdash (\!|y \cdot (x-1)! = x_0! \land x - 1 \geq 0|\!)\, \texttt{x := x - 1}\, (\!|y \cdot x! = x_0! \land x \geq 0|\!)}$$

- only creative step: invention of loop invariant $y \cdot x! = x_0! \land x \geq 0$
- quite unreadable, introduce proof tableaux



# Proof Tableaux



## Problems in Presentation of Hoare Calculus

- proof trees become quite large even for small examples
- reason: lots of duplication, e.g., in composition rule

$$\dfrac{\vdash (\!|\varphi|\!)\, C_1\, (\!|\eta|\!) \qquad \vdash (\!|\eta|\!)\, C_2\, (\!|\psi|\!)}{\vdash (\!|\varphi|\!)\, C_1; C_2\, (\!|\psi|\!)} \text{ composition}$$

  every formula $\varphi$, $\eta$, $\psi$ occurs twice
- aim: develop better representation of Hoare-calculus proofs





## Proof Tableaux

- main ideas
  - write program commands line-by-line
  - interleave program commands with midconditions
- structure

$$\begin{array}{l} \quad (\!|\varphi_0|\!) \\ C_1; \\ \quad (\!|\varphi_1|\!) \\ C_2; \\ \quad (\!|\varphi_2|\!) \\ \dots \\ C_n \\ \quad (\!|\varphi_n|\!) \end{array}$$

  where none of the $C_i$ is a sequential execution
- idea: each midcondition $\varphi_i$ should hold after execution of $C_i$

## Weakest Preconditions

$$
\begin{array}{l}
\qquad (\!|\varphi_i|\!) \\
C_{i+1}; \\
\qquad (\!|\varphi_{i+1}|\!)
\end{array}
$$

- problem: how to find all the midconditions $\varphi_i$?
- solution
  - assume $\varphi_{i+1}$ (and of course $C_{i+1}$) is given
  - then try to compute $\varphi_i$ as weakest precondition, i.e., $\varphi_i$ should be logically weakest formula satisfying

$$\models (\!|\varphi_i|\!)\, C_i\, (\!|\varphi_{i+1}|\!)$$

  - we will see, that such weakest preconditions can for many commands be computed automatically





## Constructing the Proof Tableau

- aim: verify $\vdash (\!|\varphi_0'|\!)\, C_1; \dots; C_n\, (\!|\varphi_n|\!)$
- approach: compute formulas $\varphi_{n-1}, \dots, \varphi_0$, e.g., by taking weakest preconditions

$$
\begin{array}{l}
\qquad (\!|\varphi_0|\!) \\
C_1; \\
\qquad (\!|\varphi_1|\!) \\
\quad \dots \\
C_{n-1}; \\
\qquad (\!|\varphi_{n-1}|\!) \\
C_n \\
\qquad (\!|\varphi_n|\!)
\end{array}
$$

  and check $\models \varphi_0' \longrightarrow \varphi_0$
  this last check corresponds to an application of the implication-rule
- next: consider the various commands how to compute a suitable formula $\varphi_i$ given $C_{i+1}$ and $\varphi_{i+1}$





## Constructing the Proof Tableau – Assignment

- for the assignment, the weakest precondition is computed via

$$
\begin{array}{l}
\qquad (\!|\varphi[x/e]|\!) \\
x := e \\
\qquad (\!|\varphi|\!)
\end{array}
$$

- application is completely automatic: just substitute





## Constructing the Proof Tableau – Implication

- represent implication-rule by writing two consecutive formulas

$$
\begin{array}{l}
(\!|\psi|\!) \\
(\!|\varphi|\!)
\end{array}
$$

  whenever $\models \psi \longrightarrow \varphi$
- application
  - simplify formulas
  - close proof tableau at the top, to turn given precondition into computed formula at top of program, e.g., $\models \varphi_0' \longrightarrow \varphi$ on slide 22
- example proof of $\vdash (\!|y = 2|\!)$ `y := y * y; x := y + 1` $(\!|x = 5|\!)$

$$
\begin{array}{lll}
 & (\!|y = 2|\!) & \\
 & (\!|y \cdot y = 4|\!) & \text{(closing proof tableau at top)} \\
\texttt{y := y * y} & & \\
 & (\!|y = 4|\!) & \text{(optional simplification step)} \\
 & (\!|y + 1 = 5|\!) & \\
\texttt{x := y + 1} & & \\
 & (\!|x = 5|\!) &
\end{array}
$$

## Example with Destructive Updates

- assume we want to calculate $u = x + y$ via the following program $P$

$$
\begin{array}{ll}
 & (\!|\mathsf{true}|\!) \\
 & (\!|x + y = x + y|\!) \\
\texttt{z := x} & \\
 & (\!|z + y = x + y|\!) \\
\texttt{z := z + y} & \\
 & (\!|z = x + y|\!) \\
\texttt{u := z} & \\
 & (\!|u = x + y|\!)
\end{array}
$$

- the midconditions have been inserted fully automatic
- hence we easily conclude $\vdash (\!|\mathsf{true}|\!)\ P\ (\!|u = x + y|\!)$
- note: although the tableau is constructed bottom-up, it also makes sense to read it top-down





## An Invalid Example

- consider the following invalid tableau

$$
\begin{array}{ll}
 & (\!|\mathsf{true}|\!) \\
 & (\!|x + 1 = x + 1|\!) \\
\texttt{x := x + 1} & \\
 & (\!|x = x + 1|\!)
\end{array}
$$

- if the tableau were okay, then the result would be the arithmetic property $x = x + 1$, a formula that does not hold for any number $x$
- problem in tableau
  - assignment rule was not applied correctly
  - reason: substitution has to replace all variables
- corrected version

$$
\begin{array}{ll}
 & (\!|x + 1 = (x + 1) + 1|\!) \\
\texttt{x := x + 1} & \\
 & (\!|x = x + 1|\!)
\end{array}
$$





## Constructing the Proof Tableau – If-Then-Else

- aim: calculate $\varphi$ such that

$$\vdash (\!|\varphi|\!)\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ (\!|\psi|\!)$$

  can be derived
- applying our procedure recursively, we get
  - formula $\varphi_1$ such that $\vdash (\!|\varphi_1|\!)\ C_1\ (\!|\psi|\!)$ is derivable
  - formula $\varphi_2$ such that $\vdash (\!|\varphi_2|\!)\ C_2\ (\!|\psi|\!)$ is derivable
- then weakest precondition for if-then-else is formula

$$\varphi := (b \longrightarrow \varphi_1) \wedge (\neg b \longrightarrow \varphi_2)$$

- formal justification that $\varphi$ is sound

$$\dfrac{\dfrac{\vdash (\!|\varphi_1|\!)\ C_1\ (\!|\psi|\!)}{\vdash (\!|\varphi \wedge b|\!)\ C_1\ (\!|\psi|\!)} \quad \dfrac{\vdash (\!|\varphi_2|\!)\ C_2\ (\!|\psi|\!)}{\vdash (\!|\varphi \wedge \neg b|\!)\ C_2\ (\!|\psi|\!)}}{\vdash (\!|\varphi|\!)\ \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ (\!|\psi|\!)}$$





## Example with If-Then-Else

- consider non-optimal code to compute the successor

$$
\begin{array}{lll}
 & (\!|\mathsf{true}|\!) & \\
 & (\!|((x+1) - 1 = 0 \longrightarrow 1 = x + 1) \wedge ((x+1) - 1 \neq 0 \longrightarrow x + 1 = x + 1)|\!) & \\
\texttt{a := x + 1;} & & \\
 & (\!|(a - 1 = 0 \longrightarrow 1 = x + 1) \wedge (a - 1 \neq 0 \longrightarrow a = x + 1)|\!) & \\
\texttt{if (a - 1 = 0) then \{} & & \\
 & (\!|1 = x + 1|\!) & \\
\quad \texttt{y := 1} & & \\
 & (\!|y = x + 1|\!) & \text{(formula copied to end of then-branch)} \\
\texttt{\} else \{} & & \\
 & (\!|a = x + 1|\!) & \\
\quad \texttt{y := a} & & \\
 & (\!|y = x + 1|\!) & \text{(formula copied to end of else-branch)} \\
\texttt{\}} & & \\
 & (\!|y = x + 1|\!) &
\end{array}
$$

- insertion of midconditions is completely automatic
- large formula obtained in 2nd line must be proven in underlying logic

## Applying the While Rule

$$\frac{\vdash (\!|\eta \wedge b|\!)\ C\ (\!|\eta|\!)}{\vdash (\!|\eta|\!)\ \mathtt{while}\ b\ C\ (\!|\eta \wedge \neg b|\!)}\ \text{while}$$

- let us consider applicability in combination with implication-rule for arbitrary setting: how to derive the following?
$$\vdash (\!|\varphi|\!)\ \mathtt{while}\ b\ C\ (\!|\psi|\!)$$
solution: find invariant $\eta$ such that
  - $\models \varphi \longrightarrow \eta$ — precondition implies invariant
  - $\vdash (\!|\gamma|\!)\ C\ (\!|\eta|\!)$ — handle loop body recursively, produces $\gamma$
  - $\models \eta \wedge b \longrightarrow \gamma$ — $\eta$ is indeed invariant
  - $\models \eta \wedge \neg b \longrightarrow \psi$ — invariant and $\neg b$ implies postcondition
- notes
  - invariant $\eta$ has to be satisfied at beginning and end of loop-body, but not in between
  - invariant often captures the core of an algorithm: it describes connection between variables throughout execution
  - finding invariant is not automatic, but for seeing the connection it often helps to execute the loop a few rounds





## Applying the While Rule – Soundness

$$\frac{\vdash (\!|\eta \wedge b|\!)\ C\ (\!|\eta|\!)}{\vdash (\!|\eta|\!)\ \mathtt{while}\ b\ C\ (\!|\eta \wedge \neg b|\!)}\ \text{while}$$

- let us consider applicability in combination with implication-rule for arbitrary setting: how to derive the following?
$$\vdash (\!|\varphi|\!)\ \mathtt{while}\ b\ C\ (\!|\psi|\!)$$
solution: find invariant $\eta$ such that
  - $\models \varphi \longrightarrow \eta$ — precondition implies invariant
  - $\vdash (\!|\gamma|\!)\ C\ (\!|\eta|\!)$ — handle loop body recursively, produces $\gamma$
  - $\models \eta \wedge b \longrightarrow \gamma$ — $\eta$ is indeed invariant
  - $\models \eta \wedge \neg b \longrightarrow \psi$ — invariant and $\neg b$ implies postcondition
- soundness proof

$$\cfrac{\cfrac{\cfrac{\vdash (\!|\gamma|\!)\ C\ (\!|\eta|\!)}{\vdash (\!|\eta \wedge b|\!)\ C\ (\!|\eta|\!)}}{\vdash (\!|\eta|\!)\ \mathtt{while}\ b\ C\ (\!|\eta \wedge \neg b|\!)}}{\vdash (\!|\varphi|\!)\ \mathtt{while}\ b\ C\ (\!|\psi|\!)}$$





## Schema to Find Loop Invariant

- to create a Hoare-triple for a while-loop
$$\vdash (\!|\varphi|\!)\ \mathtt{while}\ b\ C\ (\!|\psi|\!)$$
find $\eta$ such that
  - $\models \varphi \longrightarrow \eta$ — precondition implies invariant
  - $\vdash (\!|\gamma|\!)\ C\ (\!|\eta|\!)$ — handle loop body recursively, produces $\gamma$
  - $\models \eta \wedge b \longrightarrow \gamma$ — $\eta$ is invariant
  - $\models \eta \wedge \neg b \longrightarrow \psi$ — invariant and $\neg b$ implies postcondition
- approach to find $\eta$
  1. guess initial $\eta$, e.g., based on a few loop executions
  2. check $\models \varphi \longrightarrow \eta$ and $\models \eta \wedge \neg b \longrightarrow \psi$; if not successful modify $\eta$
  3. compute $\gamma$ by bottom-up generation of $\vdash (\!|\gamma|\!)\ C\ (\!|\eta|\!)$
  4. check $\models \eta \wedge b \longrightarrow \gamma$
  5. if last check is successful, proof is done
  6. otherwise, adjust $\eta$
- note: if $\varphi$ is not known for checking $\models \varphi \longrightarrow \eta$, then instead perform bottom-up propagation of commands before while-loop (starting with $\eta$) and then use precondition of whole program





## Verification of Factorial Program – Initial Invariant

- program $P$: `y := 1; while x > 0 {y := y * x; x := x - 1}`
- aim: $\vdash (\!|x = x_0 \wedge x \geq 0|\!)\ P\ (\!|y = x_0!|\!)$
- for guessing initial invariant, execute a few iterations to compute $6!$

| iteration | $x_0$ | $x$ | $y$ | $x!$ |
|---|---|---|---|---|
| 0 | 6 | 6 | 1 | 720 |
| 1 | 6 | 5 | 6 | 120 |
| 2 | 6 | 4 | 30 | 24 |
| 3 | 6 | 3 | 120 | 6 |
| 4 | 6 | 2 | 360 | 2 |
| 5 | 6 | 1 | 720 | 1 |

  observations
  - column $x!$ was added since computing $x!$ is aim
  - multiplication of $y$ and $x!$ stays identical: $y \cdot x! = x_0!$
  - hence use $y \cdot x! = x_0!$ as initial candidate of invariant
- alternative reasoning with symbolic execution
  - in $y$ we store $x_0 \cdot (x_0 - 1) \cdot \ldots \cdot (x + 1) = x_0!/x!$, so multiplying with $x!$ we get $y \cdot x! = x_0!$

## Verification of Factorial Program – Testing Initial Invariant

- initial invariant: $\eta = (y \cdot x! = x_0!)$
- potential proof tableau

| | | |
|---|---|---|
| | $(\!|x = x_0 \wedge x \geq 0|\!)$ | |
| | $(\!|1 \cdot x! = x_0!|\!)$ | (implication verified) |
| `y := 1;` | | |
| | $(\!|\eta|\!)$ | |
| `while (x > 0) {` | | |
| | $(\!|\eta \wedge x > 0|\!)$ | |
| `  y := y * x;` | | |
| `  x := x - 1` | | |
| | $(\!|\eta|\!)$ | |
| `}` | | |
| | $(\!|\eta \wedge \neg x > 0|\!)$ | |
| | $(\!|y = x_0!|\!)$ | (implication does not hold) |

- problem: condition $\neg x > 0$ ($x \leq 0$) does not enforce $x = 0$ at end





## Verification of Factorial Program – Strengthening Invariant

- strengthened invariant: $\eta = (y \cdot x! = x_0! \wedge x \geq 0)$
- potential proof tableau

| | | |
|---|---|---|
| | $(\!|x = x_0 \wedge x \geq 0|\!)$ | |
| | $(\!|1 \cdot x! = x_0! \wedge x \geq 0|\!)$ | (implication verified) |
| `y := 1;` | | |
| | $(\!|\eta|\!)$ | |
| `while (x > 0) {` | | |
| | $(\!|\eta \wedge x > 0|\!)$ | |
| | $(\!|(y \cdot x) \cdot (x-1)! = x_0! \wedge x - 1 \geq 0|\!)$ | (implication verified) |
| `  y := y * x;` | | |
| | $(\!|y \cdot (x-1)! = x_0! \wedge x - 1 \geq 0|\!)$ | |
| `  x := x - 1` | | |
| | $(\!|\eta|\!)$ | |
| `}` | | |
| | $(\!|\eta \wedge \neg x > 0|\!)$ | |
| | $(\!|y = x_0!|\!)$ | (implication verified) |

- proof completed, since all implications verified (e.g. by SMT solver)





## Larger Example – Minimal-Sum Section

- assume extension of programming language: read-only arrays (writing into arrays requires significant extension of calculus)
- user is responsible for proper array access
- problem definition
  - given array $a[0], \dots, a[n-1]$ of length $n$, a section of $a$ is a continuous block $a[i], \dots, a[j]$ with $0 \leq i \leq j < n$
  - define $S_{i,j}$ as sum of section
  $$S_{i,j} := a[i] + \cdots + a[j]$$
  - section $(i, j)$ is minimal, if $S_{i,j} \leq S_{i',j'}$ for all sections $(i', j')$ of $a$
- example: consider array $[-7, 15, -1, 3, 15, -6, 4, -5]$
  - $[3, 15, -6]$ and $[-6]$ are sections, but $[3, -6, 4]$ is not
  - there are two minimal-sum sections: $[-7]$ and $[-6, 4, -5]$





## Minimal-Sum Section – Tasks

- write a program that computes sum of minimal section
- write a specification that makes "compute sum of minimal section" formal
- show that program satisfies the formal specification

## Minimal-Sum Section – Challenges

- trivial algorithm
  - compute all sections ($O(n^2)$)
  - compute all sums of these sections and find the minimum
  - results in $O(n^3)$ algorithm
- aim: $O(n)$-algorithm which reads the array only once
- consequence: proof required that it is not necessary to explicitly compute all $O(n^2)$ sections
- example: consider array $[-8, 3, -65, 20, 45, -100, -8, 17, -4, -14]$
  - when reading from left-to-right a promising candidate might be $[-8, 3, -65]$, but there also is the later $[-100, -8]$, so how to decide what to take?





## Minimal-Sum Section – Algorithm

- idea of algorithm
  - $k$: index that traverses array from left-to-right
  - $s$: minimal-sum of all sections seen so far
  - $t$: minimal-sum of all sections that end at position $k - 1$
- algorithm *Min_Sum*

```
k := 1;
t := a[0];
s := a[0];
while (k != n) {
  t := min(t + a[k], a[k]);
  s := min(s, t);
  k := k + 1
}
```

- correctness not obvious, so let us better prove it





## Minimal-Sum Section – Specification

- we split the specification in two parts via two Hoare-triples
  - $Sp_1$ specifies that the value of $s$ is smaller than the sum of any section

$$(\!|\text{true}|\!)\ \textit{Min\_Sum}\ (\!|\forall i, j.\ 0 \leq i \leq j < n \longrightarrow s \leq S_{i,j}|\!)$$

  - $Sp_2$ specifies that there exists some section whose sum is $s$

$$(\!|\text{true}|\!)\ \textit{Min\_Sum}\ (\!|\exists i, j.\ 0 \leq i \leq j < n \wedge s = S_{i,j}|\!)$$





## Minimal-Sum Section – Proving $Sp_1$

```
k := 1;
t := a[0];
s := a[0];
while (k != n) {
  t := min(t + a[k], a[k]);
  s := min(s, t);
  k := k + 1
}
```

$$Sp_1 : (\!|\text{true}|\!)\ \textit{Min\_Sum}\ (\!|\forall i, j.\ 0 \leq i \leq j < n \longrightarrow s \leq S_{i,j}|\!)$$

- find candidate invariant
  - invariant often similar to postcondition
  - invariant expresses relationships that are valid at beginning of each loop-iteration
- suitable invariant is $Inv_1(s, k)$ defined as

$$\forall i, j.\ 0 \leq i \leq j < k \longrightarrow s \leq S_{i,j}$$

$(\!| \mathit{Inv}_1(a[0],1) |\!)$ (true statement)

```
k := 1;
```
$(\!| \mathit{Inv}_1(a[0],k) |\!)$

```
t := a[0];
```
$(\!| \mathit{Inv}_1(a[0],k) |\!)$

```
s := a[0];
```
$(\!| \mathit{Inv}_1(s,k) |\!)$

```
while (k != n) {
```
$(\!| \mathit{Inv}_1(s,k) \wedge k \neq n |\!)$

$(\!| \mathit{Inv}_1(\min(s, \min(t + a[k], a[k])), k+1) |\!)$ (does not hold, no info on $t$)

```
  t := min(t + a[k], a[k]);
```
$(\!| \mathit{Inv}_1(\min(s,t), k+1) |\!)$

```
  s := min(s, t);
```
$(\!| \mathit{Inv}_1(s, k+1) |\!)$

```
  k := k + 1;
```
$(\!| \mathit{Inv}_1(s,k) |\!)$

```
}
```
$(\!| \mathit{Inv}_1(s,k) \wedge \neg k \neq n |\!)$

$(\!| \mathit{Inv}_1(s,n) |\!)$ (implication verified)





## Minimal-Sum Section – Strengthening Invariant

```
k := 1;
t := a[0];
s := a[0];
while (k != n) {
  t := min(t + a[k], a[k]);
  s := min(s, t);
  k := k + 1
}
```

$$Sp_1 : (\!|\text{true}|\!)\ \mathit{Min\_Sum}\ (\!|\forall i,j.\ 0 \le i \le j < n \longrightarrow s \le S_{i,j}|\!)$$

- suitable invariant for $s$ is $Inv_1(s,k)$ defined as

$$\forall i,j.\ 0 \le i \le j < k \longrightarrow s \le S_{i,j}$$

- define similar invariant for $t$: $Inv_2(t,k)$ defined as

$$\forall i.\ 0 \le i < k \longrightarrow t \le S_{i,k-1}$$

- now try strengthened invariant $Inv_1(s,k) \wedge Inv_2(t,k)$





$(\!| \mathit{Inv}_1(a[0],1) \wedge \mathit{Inv}_2(a[0],1) |\!)$ (true statement)

```
k := 1;
```
$(\!| \mathit{Inv}_1(a[0],k) \wedge \mathit{Inv}_2(a[0],k) |\!)$

```
t := a[0];
```
$(\!| \mathit{Inv}_1(a[0],k) \wedge \mathit{Inv}_2(t,k) |\!)$

```
s := a[0];
```
$(\!| \mathit{Inv}_1(s,k) \wedge \mathit{Inv}_2(t,k) |\!)$

```
while (k != n) {
```
$(\!| \mathit{Inv}_1(s,k) \wedge \mathit{Inv}_2(t,k) \wedge k \neq n |\!)$

$(\!| \mathit{Inv}_1(\min(s, \min(t + a[k], a[k])), k+1) \wedge \mathit{Inv}_2(\min(t + a[k], a[k]), k+1) |\!)$ (implication verified)

```
  t := min(t + a[k], a[k]);
```
$(\!| \mathit{Inv}_1(\min(s,t), k+1) \wedge \mathit{Inv}_2(t, k+1) |\!)$

```
  s := min(s, t);
```
$(\!| \mathit{Inv}_1(s, k+1) \wedge \mathit{Inv}_2(t, k+1) |\!)$

```
  k := k + 1;
```
$(\!| \mathit{Inv}_1(s,k) \wedge \mathit{Inv}_2(t,k) |\!)$

```
}
```
$(\!| \mathit{Inv}_1(s,k) \wedge \mathit{Inv}_2(t,k) \wedge \neg k \neq n |\!)$

$(\!| \mathit{Inv}_1(s,n) |\!)$ (implication verified)





## Minimal-Sum Section – Proving the Implications

- invariants
  - $Inv_1(s,k) := \forall i,j.\ 0 \le i \le j < k \longrightarrow s \le S_{i,j}$
  - $Inv_2(t,k) := \forall i.\ 0 \le i < k \longrightarrow t \le S_{i,k-1}$
- implications
  - true $\longrightarrow \mathit{Inv}_1(a[0],1) \wedge \mathit{Inv}_2(a[0],1)$
    - because of the conditions of the quantifiers, by fixing $k = 1$ we only have to consider section $(0,0)$, i.e, we show $a[0] \le S_{0,0} = a[0]$
  - let $0 < k < n$ where $n$ is length of array $a$; then $\mathit{Inv}_1(s,k) \wedge \mathit{Inv}_2(t,k) \wedge k \neq n$ implies both $\mathit{Inv}_2(\min(t + a[k], a[k]), k+1)$ and $\mathit{Inv}_1(\min(s, \min(t + a[k], a[k])), k+1)$;
    proof
    - pick any $0 \le i < k+1$; we show $\min(t + a[k], a[k])) \le S_{i,k}$; if $i < k$ then $S_{i,k} = S_{i,k-1} + a[k]$, so we use $\mathit{Inv}_2(t,k)$ to get $t \le S_{i,k-1}$ and thus $\min(t + a[k], a[k])) \le t + a[k] \le S_{i,k-1} + a[k] = S_{i,k}$;
      otherwise, $i = k$ and we have $\min(t + a[k], a[k]) \le a[k] = S_{i,k}$
    - pick any $0 \le i \le j < k+1$;
      we need to show $\min(s, \min(t + a[k], a[k])) \le S_{i,j}$;
      if $j = k$ then the result follows from the previous statement;
      otherwise $j < k$ and the result follows from $\mathit{Inv}_1(s,k)$

## Proof Tableaux – Summary

- we have proven soundness of non-trivial algorithm *Min_Sum*
- with gaps
  - we only proved $Sp_1$, but not $Sp_2$
  - lemma on previous slide demanded $0 < k < n$ which does not follow from loop-condition $k \neq n$; a proper fix would require a strengthened invariant which includes bounds on $k$
- main reasoning (proving the implications on previous slide) was done purely in logic with no reference to program



# Verification Condition Generation



## Verification Condition Generator – VCG

- previous part: find suitable invariants, propagate formulas via substitution, etc., and check implication
- a VCG automates this process
  - input is annotated program with pre- and post-conditions and invariants
  - VCG generates implications automatically (verification conditions)
  - verification conditions are passed to SMT-solver, theorem prover, etc., to finally show correctness
- VCG simplifies reasoning a lot, more automation
- still: in case SMT-solver or theorem prover fails, user needs to understand failure to adapt invariants, assertions, or perform manual proofs, etc.





## IMP2 – A VCG Implementation in Isabelle

- IMP2 is an Isabelle formalization of our small imperative language while-language
- it includes a VCG; the verification conditions have to be proven using the Isabelle system
  - the conditions can be forwarded to an SMT solver (`sledgehammer`)
  - the conditions can be solved by Isabelle's automation (`simp`, ...)
  - manual proofs are possible
- example: `Demo06.thy`
  (note that $x_0$ in IMP2 refers to initial value of $x$ without explicit assumption $x_0 = x$)

```
procedure_spec (partial) fact_prog(x)
  returns y
  assumes ‹x ≥ 0›
    ensures "y = fact_int x₀"
  defines ‹
    y = 1;
    while (x > 0)
      @invariant ‹y * fact_int x = fact_int x₀›
    {
      y = y * x;
      x = x - 1
    }
  ›
  apply vcg
    apply simp
   apply simp
  oops
```

## Modular Verification via VCGs

- specifications via Hoare-triple $(\!|\varphi|\!)\ P\ (\!|\psi|\!)$ may be seen as a contract between supplier and consumer of program $P$
  - supplier insists that consumer invokes $P$ only on states satisfying $\varphi$
  - supplier promises that after execution of $P$ formula $\psi$ holds
- validation of Hoare-triples with Hoare-calculus can be seen as validation of contracts for method- or procedure-calls
- once contracts have been verified for algorithm $f$, during verification one can replace program invocations of $f$ by its contract





## Example

- assume we want to write method for binomial coefficients

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

  to compute chance of lotto-jackpot $1 : \binom{49}{6}$
- example program

```
procedure_spec (partial) binom_prog(n,k)
  returns r
  assumes ‹n ≥ 0 ∧ k ≥ 0 ∧ n ≥ k›
    ensures "r = binom_int n₀ k₀"
  defines ‹
    a = fact_prog(n);
    b = fact_prog(k);
    c = fact_prog (n - k);
    r = a / (b * c)
  ›
  apply vcg
```

- in example, we need to ensure that preconditions of factorial program are met
- generated verification conditions then use post-conditions for factorial





## Programming by Contract – Advantages

- in the same way as methods help to structure larger programs, contracts for these methods help to verify larger programs
- reason: for verifying code invoking method $m$, it suffices to look at contract of $m$ – without looking at implementation of $m$
- positive effects
  - add layer of abstraction
  - easy to change implementation of $m$ as long as contract stays identical
  - verification becomes more modular
- example: for invocation of `min` in minimal-sum section it does not matter whether
  - `min` is built-in operator which is substituted as such, or
  - `min` is user-defined method that according to the contract computes the mathematical $\min$-operation



# Termination of Imperative Programs

## Adding Termination to Calculus

- since while-loops are only source of non-termination in presented imperative language, it suffices to adjust the while-rule in the Hoare-calculus

  all other Hoare-calculus rules can be used as before
- recall: total correctness = partial correctness + termination
- previous while-rule already proved partial correctness
- only task: extend existing while-rule to additionally prove termination
- idea of ensuring termination: use variants
  - a variant (or measure) is an integer expression;
  - this integer expression strictly decreases in every loop iteration and
  - at the same time the variant stays non-negative;
  - conclusion: there cannot be infinitely many loop iterations
- in IMP2, termination is proved by omitting "(partial)" in procedure definitions, and by providing a variant for every while-loop, cf. `fact_total_prog` in `Demo06.thy`





## A While-Rule For Total Correctness

- while-rule for partial correctness

$$\frac{\vdash (\!|\varphi \wedge b|\!)\, C\, (\!|\varphi|\!)}{\vdash (\!|\varphi|\!)\, \texttt{while}\ b\ C\, (\!|\varphi \wedge \neg b|\!)} \text{ while}$$

- extended while-rule for total correctness

$$\frac{\vdash (\!|\varphi \wedge b \wedge e_0 = e \geq 0|\!)\, C\, (\!|\varphi \wedge e_0 > e \geq 0|\!)}{\vdash (\!|\varphi \wedge e \geq 0|\!)\, \texttt{while}\ b\ C\, (\!|\varphi \wedge \neg b|\!)} \text{ while-total}$$

  where
  - $e$ is variant expression with values before execution of $C$
  - $e$ is (the same) variant expression with values after execution of $C$
  - $e_0$ is fresh logical variable, used to store the value of $e$ before: $e_0 = e$
  - hence, postcondition $e_0 > e$ enforces decrease of $e$ when executing $C$
  - non-negativeness is added three times, even in precondition of while
  - $e$ is of type integer so that $SN\ \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x > y \geq 0\}$ can be used as underlying terminating relation: each loop iteration corresponds to a step $([\![e]\!]_{\alpha_{\text{before}}}, [\![e]\!]_{\alpha_{\text{after}}})$ in this relation





## Applying While-Total

$$\frac{\vdash (\!|\varphi \wedge b \wedge e_0 = e \geq 0|\!)\, C\, (\!|\varphi \wedge e_0 > e \geq 0|\!)}{\vdash (\!|\varphi \wedge e \geq 0|\!)\, \texttt{while}\ b\ C\, (\!|\varphi \wedge \neg b|\!)} \text{ while-total}$$

- application
  - $e_0$ is fresh logical variable, so nothing to choose
  - variant $e$ has to be chosen, but this is often easy
    - `while (x < 5) { ... x := x + 1 ...}` is same as `while (5 - x > 0) { ... x := x + 1 ...}`, so $e = 5 - x$
    - `while (y >= x) { ... y := y - 2 ...}` is same as `while (y - x >= 0) { ... y := y - 2 ...}`, so $e = y - x\ (+2)$
    - `while (x != y) { ... y := y + 1 ...}` is same as `while (x - y != 0) { ... y := y + 1 ...}`, so $e = x - y$
  - checking the condition is then easily possible via proof tableau, in the same way as for the while-rule for partial correctness
  - all side-conditions $e \geq 0$ can completely be eliminated by choosing $e = \max(0, e')$ for some $e'$, but then proving $e_0 > e$ will become harder as it has to deal with $\max$
  - invariant $\varphi$ can be taken unchanged from partial correctness proof





## Total Correctness of Factorial Program

- red parts have been added for termination proof with variant $x - z$

| | | |
|---|---|---|
| | $(\!|\text{true} \wedge x \geq 0|\!)$ | (new termination condition on $x$) |
| | $(\!|1 = 0! \wedge x - 0 \geq 0|\!)$ | |
| `y := 1;` | | |
| | $(\!|y = 0! \wedge x - 0 \geq 0|\!)$ | |
| `z := 0;` | | |
| | $(\!|y = z! \wedge x - z \geq 0|\!)$ | (new condition added) |
| `while (x != z) {` | | |
| | $(\!|y = z! \wedge x \neq z \wedge e_0 = x - z \geq 0|\!)$ | (new condition added) |
| | $(\!|y \cdot (z+1) = (z+1)! \wedge e_0 > x - (z+1) \geq 0|\!)$ | (more reasoning) |
| `z := z + 1;` | | |
| | $(\!|y \cdot z = z! \wedge e_0 > x - z \geq 0|\!)$ | |
| `y := y * z;` | | |
| | $(\!|y = z! \wedge e_0 > x - z \geq 0|\!)$ | (new condition added) |
| `}` | | |
| | $(\!|y = z! \wedge \neg x \neq z|\!)$ | |
| | $(\!|y = x!|\!)$ | |

## Remarks on Total Correctness of Factorial Program

- precondition $x \geq 0$ was added automatically from termination proof
- in fact, the program does not terminate on negative inputs
- for factorial program (and other imperative programs) Hoare-calculus permits to prove local termination, i.e., termination on certain inputs
- in contrast, for functional program we always considered universal termination, i.e., termination of all inputs
- termination proofs can also be performed stand-alone (without partial correctness proof): just prove postcondition "true" with while-total-rule:

$$\vdash (\!|\varphi|\!)\, P\, (\!|\mathsf{true}|\!)$$

implies termination of $P$ on inputs that satisfy $\varphi$, so

$$\vdash (\!|\mathsf{true}|\!)\, P\, (\!|\mathsf{true}|\!)$$

shows universal termination of $P$



## Soundness of Hoare-Calculus



## Soundness of Hoare-Calculus

- so far, we have two notions of soundness
  - $\models (\!|\varphi|\!)\, P\, (\!|\psi|\!)$: via semantic of imperative programs, i.e., whenever $\alpha \models \varphi$ and $(P, \alpha) \hookrightarrow^* (\texttt{skip}, \beta)$ then $\beta \models \psi$ must hold
  - $\vdash (\!|\varphi|\!)\, P\, (\!|\psi|\!)$: syntactic, what can be derived via Hoare-calculus rules
- missing: soundness of calculus, i.e.,

$$\vdash (\!|\varphi|\!)\, P\, (\!|\psi|\!) \text{ implies } \models (\!|\varphi|\!)\, P\, (\!|\psi|\!)$$

- formal proof is based on big-step semantics $\to$ (see exercises): $(P, \alpha) \hookrightarrow^* (\texttt{skip}, \beta)$ is turned into $(P, \alpha) \to \beta$
- soundness of the calculus is then established by the following property, which is proven by induction w.r.t. the Hoare-calculus rules for arbitrary $\alpha, \beta$:

$$\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!) \longrightarrow \alpha \models \varphi \longrightarrow (C, \alpha) \to \beta \longrightarrow \beta \models \psi$$





## Proving $\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!) \longrightarrow \alpha \models \varphi \longrightarrow (C, \alpha) \to \beta \longrightarrow \beta \models \psi$

Case 1: implication-rule

$\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!)$ since $\models \varphi \longrightarrow \varphi'$, $\vdash (\!|\varphi'|\!)\, C\, (\!|\psi'|\!)$, and $\models \psi' \longrightarrow \psi$

- IH: $\forall \alpha, \beta.\, \alpha \models \varphi' \longrightarrow (C, \alpha) \to \beta \longrightarrow \beta \models \psi'$
- assume $\alpha \models \varphi$ and $(C, \alpha) \to \beta$
- then by $\models \varphi \longrightarrow \varphi'$ conclude $\alpha \models \varphi'$
- in combination with IH get $\beta \models \psi'$
- with $\models \psi' \longrightarrow \psi$ conclude $\beta \models \psi$

## Proving $\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!) \longrightarrow \alpha \models \varphi \longrightarrow (C,\alpha) \to \beta \longrightarrow \beta \models \psi$

Case 2: composition-rule

$\vdash (\!|\varphi|\!)\, C_1; C_2\, (\!|\psi|\!)$ since $\vdash (\!|\varphi|\!)\, C_1\, (\!|\eta|\!)$ and $\vdash (\!|\eta|\!)\, C_2\, (\!|\psi|\!)$

- IH-1: $\forall \alpha, \beta.\, \alpha \models \varphi \longrightarrow (C_1, \alpha) \to \beta \longrightarrow \beta \models \eta$
- IH-2: $\forall \alpha, \beta.\, \alpha \models \eta \longrightarrow (C_2, \alpha) \to \beta \longrightarrow \beta \models \psi$
- assume $\alpha \models \varphi$ and $(C_1; C_2, \alpha) \to \beta$
- from the latter and the definition of $\to$, there must be $\gamma$ such that $(C_1, \alpha) \to \gamma$ and $(C_2, \gamma) \to \beta$
- by using IH-1 (choose $\alpha$ and $\gamma$ in $\forall$), obtain $\gamma \models \eta$
- by using IH-2 (choose $\gamma$ and $\beta$ in $\forall$), obtain $\beta \models \psi$





## Proving $\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!) \longrightarrow \alpha \models \varphi \longrightarrow (C,\alpha) \to \beta \longrightarrow \beta \models \psi$

Case 3: if-then-else-rule

$\vdash (\!|\varphi|\!)\, \texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2\, (\!|\psi|\!)$

since $\vdash (\!|\varphi \wedge b|\!)\, C_1\, (\!|\psi|\!)$ and $\vdash (\!|\varphi \wedge \neg b|\!)\, C_2\, (\!|\psi|\!)$

- IH-1: $\forall \alpha, \beta.\, \alpha \models \varphi \wedge b \longrightarrow (C_1, \alpha) \to \beta \longrightarrow \beta \models \psi$
- IH-2: $\forall \alpha, \beta.\, \alpha \models \varphi \wedge \neg b \longrightarrow (C_2, \alpha) \to \beta \longrightarrow \beta \models \psi$
- assume $\alpha \models \varphi$ and $(\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2, \alpha) \to \beta$
- perform case analysis on $[\![b]\!]_\alpha$
- w.l.o.g. we only consider the case $[\![b]\!]_\alpha =$ true where
  - from $\alpha \models \varphi$ conclude $\alpha \models \varphi \wedge b$
  - from $(\texttt{if}\ b\ \texttt{then}\ C_1\ \texttt{else}\ C_2, \alpha) \to \beta$ conclude $(C_1, \alpha) \to \beta$
  - by using IH-1 get $\beta \models \psi$





## Proving $\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!) \longrightarrow \alpha \models \varphi \longrightarrow (C,\alpha) \to \beta \longrightarrow \beta \models \psi$

Case 4: assignment-rule

$\vdash (\!|\varphi|\!)\, x := e\, (\!|\psi|\!)$ since $\varphi = \psi[x/e]$

- assume $\alpha \models \varphi$ and $(x := e, \alpha) \to \beta$
- by definition of $\to$, conclude $\beta = \alpha[x := [\![e]\!]_\alpha]$
- hence assumption $\alpha \models \varphi$ is equivalent to
  - $\alpha \models \psi[x/e]$ by unrolling $\varphi$-equality
  - $\alpha[x := [\![e]\!]_\alpha] \models \psi$ by substitution lemma for formulas
  - $\beta \models \psi$ by unrolling $\beta$-equality





## Proving $\vdash (\!|\varphi|\!)\, C\, (\!|\psi|\!) \longrightarrow \alpha \models \varphi \longrightarrow (C,\alpha) \to \beta \longrightarrow \beta \models \psi$

Case 5: while-rule

$\vdash (\!|\varphi|\!)\, \texttt{while}\ b\ C'\, (\!|\psi|\!)$ since $\vdash (\!|\varphi \wedge b|\!)\, C'\, (\!|\varphi|\!)$ and $\psi = \varphi \wedge \neg b$

- (outer) IH: $\forall \alpha, \beta.\, \alpha \models \varphi \wedge b \longrightarrow (C', \alpha) \to \beta \longrightarrow \beta \models \varphi$
- we now prove $\alpha \models \varphi \longrightarrow (\texttt{while}\ b\ C', \alpha) \to \beta \longrightarrow \beta \models \psi$ by an inner induction on $\alpha$ w.r.t. $\to$, but for fixed $b$, $C'$, $\beta$, $\varphi$, $\psi$
  - case 1: $(\texttt{while}\ b\ C', \alpha) \to \beta$ since $[\![b]\!]_\alpha =$ false and $\beta = \alpha$
    - in this case conclude $\beta = \alpha \models \varphi \wedge \neg b = \psi$
  - case 2: $(\texttt{while}\ b\ C', \alpha) \to \beta$ since $[\![b]\!]_\alpha =$ true, $(C', \alpha) \to \gamma$ and $(\texttt{while}\ b\ C', \gamma) \to \beta$
    - inner IH: $\gamma \models \varphi \longrightarrow \beta \models \psi$
    - assume $\alpha \models \varphi$
    - hence $\alpha \models \varphi \wedge b$
    - by outer IH (choose $\alpha$ and $\gamma$ in $\forall$) get $\gamma \models \varphi$
    - then inner IH yields $\beta \models \psi$

## Summary of Soundness of Hoare-Calculus

- since Hoare-calculus rules and semantics are formally defined, it is possible to verify soundness of the calculus
- proof requires inner induction for while-loop,
  since big-step semantics of while-command refers to itself
- here: only soundness of Hoare-calculus for partial correctness
- possible extension: total correctness
  - define semantic notion $\models_{total} (\!|\varphi|\!)\, C\, (\!|\psi|\!)$ stating total correctness
  - prove that Hoare-calculus with while-total is sound w.r.t. $\models_{total}$



## Summary – Verification of Imperative Programs

- covered
  - syntax and semantic of small imperative programming language
  - Hoare-calculus to verify Hoare-triples $(\!|\varphi|\!)\, P\, (\!|\psi|\!)$
  - proof tableaux and automation:
    use VCG that converts program logic into implications (verification conditions) that must be shown in underlying logic
  - proofs are often automatic, given good pre- and post-conditions and (in)variants
  - soundness of Hoare-calculus
  - programming by contracts: abstract from concrete method-implementations, use contracts
  - example VCG for core language: IMP2, based on Isabelle
- not covered
  - heap-access, references, arrays, etc.: extension to separation logic, memory model
  - bounded integers: reasoning engine for bit-vector-arithmetic
  - multi-threading