Software Engineering

GINF41E7

Olivier Gruber

Lydie du Bousquet

Frédéric Lang

Software Engineering – Week 10
Verification using formal methods

Part I:
Proving sequential programs correct

Code should be « **working** » not only « **running** »

- This is why testing was introduced (week 4)
- Testing is good and necessary, but it has limitations

Limitation of testing #1

- Required test coverage may be out of reach

    too many lines of code,
    too many branches,
    parallelism,
    …

- The probability of rare bugs to occur during testing is far below the probability that they occur during the software life

Limitation of testing #2

- Tested program may be nondeterministic (same inputs $\Rightarrow$ different outputs)
- Several possible causes:
  - Variable testing environment
    e.g., compiler, architecture, load, …
  - Programming errors
    e.g., use of non-initialized variable, div-by-0, …
  - Intrinsic nondeterminism
    e.g., parallel systems (variable communication delays, asynchrony)

Example (1/3)

Test the following C program

```
int main () {
        int x = 1;
        x = x++;
        assert (x == 2);
}
```

Example (2/3)

- Tested on Linux iX86 with Gnu CC 4.4.5 compiler: test passes
- Test is exhaustive and successful!
- Program can thus be safely deployed in customer environment

Example (3/3)

- Customer uses 32-bit SunCC/Solaris compiler
- Assertion is violated: x == 1
- Bug due to ambiguous definition of `x = x++`
  Unspecified order between assignment « `x = ...` » and increment « `x++` »

```
R = x; x = R + 1; x = R; /* x == 1 */
```

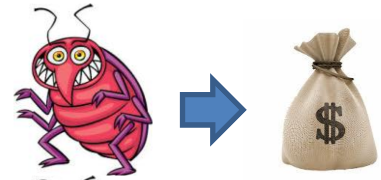
vs.

```
R = x; x = R; x = R + 1; /* x == 2 */
```

where `R` is a register used to store the initial value of `x`

Problem

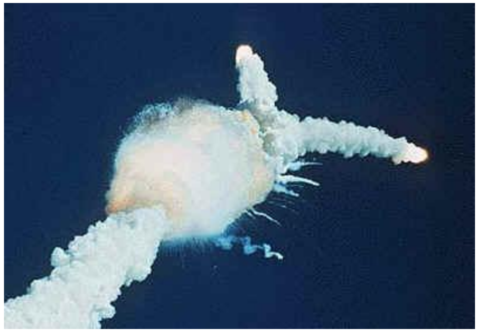
- Errors have a cost that increases over time

« Normal » software: 

Critical software: 

e.g., avionics, aerospace, automotive, nuclear, chemicals, …

- Verification methods complementary to test are needed to find bugs early (design, implementation)



Formal verification methods

- Formal = mathematically defined
- Relies on formal languages to describe:
  - Programs
  - Requirements
- **Advantages**:
  - Eliminate the risk of ambiguities
  - Offer mathematically based (rigorous) verification methods

Formal verification

- Several formal verification methods exist, with many criteria of choice
- One major criteria is whether the program is *sequential* or *concurrent*

This lecture:

- Week 10: sequential programs – proof
- Week 11: concurrent programs – model based verification

Disclaimer

- This lecture gives only a partial view of formal methods
- Other formal verification methods include:
  - Typing
  - Static analysis by abstract interpretation
  - Etc.
- There also exist other techniques and models, but we cannot address all, this is only an introduction

Proving sequential programs

- **Goal**: ensure that program behaves *as expected*
- Several possible notions of *as expected*
  - Absence of crash: No unexpected termination
    Examples: division by zero, out-of-bound array access, etc.
  - Correctness: A particular relation between program inputs and outputs holds
  - Termination: No infinite execution
  - Performance: bounded usage of resources (e.g., time, memory, etc.)
- This lecture focuses on program correctness and a little about termination

Program correctness

- Proving a program correct requires:
  - A formal specification of the program
  - A formal specification of the mathematical property that the program should satisfy
  - Formal deduction rules to relate property and program
- This is all about reasoning on programs

Hoare logic

- A simple framework for proving programs, proposed by Tony Hoare in 1969, inspired by Robert Floyd
- Mathematical formalization of deduction rules for reasoning on programs
- Motivations:
  - Rigorous definition of reasoning (teaching, research papers, …)
  - Implementation in tools

Informal example

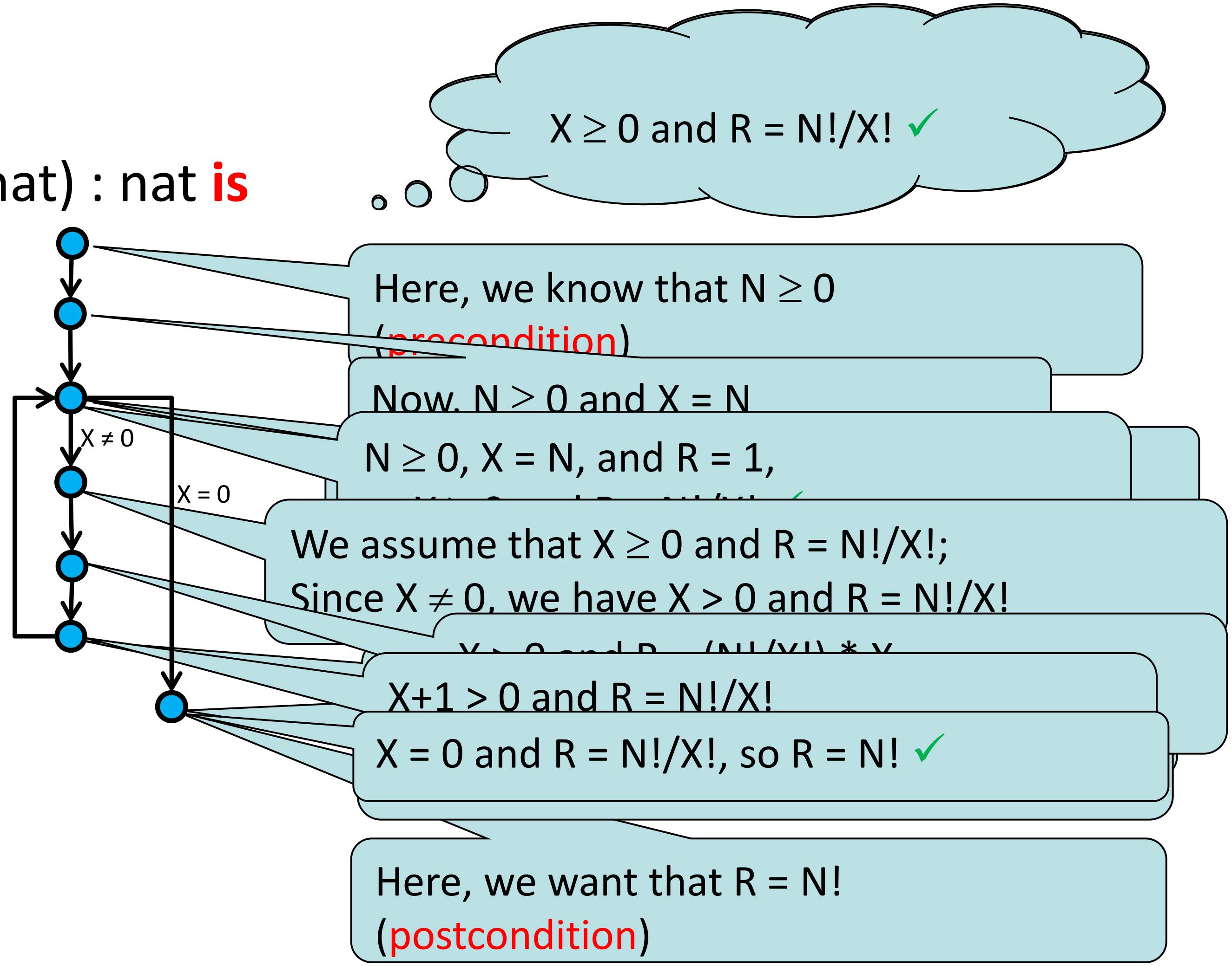
How to get convinced that the following program implements the factorial function $N! = \prod_{i \in 1..N} i$

```
function fact (N : nat) : nat is
    X := N;
    R := 1;
    while X ≠ 0 do
            R := R * X;
            X := X – 1
    end;
    return R
end
```

Reasoning at program locations

function fact (N : nat) : nat **is**

    X := N;
    R := 1;
    while X ≠ 0 **do**
        R := R * X;
        X := X – 1
    end;
    return R
end



Sequential programs

- Simple programming language without procedures
- Syntax:

  C ::= **skip**
  | X := E
  | C_1; C_2
  | **if** E **then** C_1 **else** C_2 **end**
  | **while** E **do** C_0 **end**

- Semantics is clear and not presented formally

  Note: **if** E **then** C_1 **end** = **if** E **then** C_1 **else skip end**

Logic properties

- Properties are described using first-order logic
- Formulas of the logic are defined as follows:
  - Every mathematical Boolean expression is a formula
    Examples: true, false, X == 1, N $\leq$ 4, etc.
  - Formulas combined using standard logic connectors ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$) are also formulas
    Examples: $\neg$true, $X \geq 0 \wedge X \neq 0 \Rightarrow X \geq 1$, etc.
  - Formulas bound using logic quantifiers ($\forall$, $\exists$) are also formulas (they won't be used in this lecture)
    Example: $(\exists Y)\ X = Y + 1$

Proving or disproving properties

- Proving a property consists in establishing that it is always true
  (we will say that the property *holds*)
- Disproving a property consists in establishing that it can be false (it does not hold)
- I assume that you have basic knowledge in logic, so that you can prove/disprove simple properties

Exercise

$X \Rightarrow Y$	Y = true	Y = false
X = true	true	false
X = false	true	true

(same as $\neg X \vee Y$)

Let X denote a natural number.

Do the following properties hold?

true	false
$X \leq 0$	$X \leq 0 \vee X > 0$
$X > 0 \Rightarrow X \geq 1$	$X \geq 1 \Rightarrow X > 0$
$X > 0 \Rightarrow X > 0 \vee Y > 0$	$X > 0 \Rightarrow X > 0 \wedge Y > 0$
$X > 0 \wedge Y > 0 \Rightarrow X > 0$	$X > 0 \vee Y > 0 \Rightarrow X > 0$
$\text{true} \Rightarrow X > 0$	$X = X{+}1 \Rightarrow X > 0$
$\text{true} \Rightarrow X = X{+}1$	$S = \emptyset \wedge X \in S \Rightarrow \lvert S \rvert = 1$

(where $\emptyset$ denotes the empty set and $|S|$ the size of set S)

Weaker/stronger property

- A property P is said to be *stronger* than a property Q if Q can be deduced from P, i.e.,

$$P \Rightarrow Q$$

- Conversely, Q is said to be *weaker* than P

Exercise: say which property is weaker and which is stronger (if any) among the following:

$X > 0$ vs. $X \geq 0$

$X = 0 \wedge Y = 0$ vs. $X \geq 0$

$P \wedge Q$ vs. P

$P \vee Q$ vs. P

$P \Rightarrow Q$ vs. P

$P \Rightarrow Q$ vs. Q

$\neg P$ vs. P

Variable substitution

- An operation on formulas needed in the sequel to deal with assignments
- Given formula P, variable X, and expression E

$$P[X := E]$$

  is the result of replacing every occurrence of X in P by E
- Example :

$$(X = 4)[X := X+1] = (X+1 = 4)$$

Hoare triple – proof goal

- A Hoare triple is written { P } C { Q }, where:
  - C is a program
  - P is a formula called precondition
  - Q is a formula called postcondition
- Meaning: If P holds before executing C, then Q holds after executing C
- Hoare logic is about proving Hoare triples; A Hoare triple that is to be proven is called a proof goal
- { P }, { Q } are also called assertions

Example

The proof goal (Hoare triple) for the factorial code:

$\{ N \geq 0 \}$

```
X := N;
R := 1;
while X ≠ 0 do
    R := R * X;
    X := X – 1
end
```

$\{ R = N! \}$

Deduction rules

- Deduction rules define how a proof goal can be transformed into simpler proof subgoals and/or properties to be proven
- The deduction rules are guided by the syntax: one rule per programming language construct
- A proof is an iterative application of the proof subgoals until we get only properties that hold

Syntax of deduction rules

$$\frac{\text{List of proof subgoals and properties}}{\text{Proof goal}}$$

Means:

To prove the proof goal, it suffices to prove all proof subgoals and properties

In the end, the set of all properties to be proven are often called the *proof obligations*

Rule of inaction

$$\frac{P \Rightarrow Q}{\{\ P\ \}\ \textbf{skip}\ \{\ Q\ \}}$$

To prove that

*If P holds before executing **skip**,*

*then Q holds after executing **skip***

one must prove the property $P \Rightarrow Q$

Exercises

Can you prove the following goals?

- $\{X > 0\}$ **skip** $\{X > 0\}$
- $\{X > 0\}$ **skip** $\{X \geq 0\}$
- $\{X \geq 0\}$ **skip** $\{X > 0\}$

Rule of variable assignment

$$\frac{P \Rightarrow Q[X := E]}{\{\ P\ \}\ X := E\ \{\ Q\ \}}$$

To prove that

If P holds before executing X := E,

then Q holds after executing X := E

one must prove the formula $P \Rightarrow Q[X := E]$

Example

The following goal holds

$$\{ \text{true} \}\ X := 2\ \{ X = 2 \}$$

Formally:

$$(X = 2)[X := 2] = (2 = 2)$$

and true $\Rightarrow$ $(2 = 2)$ holds

Exercise

Prove the following goals

- $\{X = 1\}\ X := 2\ \{X = 2\}$
- $\{X = 2\}\ X := X + 1\ \{X = 3\}$
- $\{X \geq Y\}\ X := X + 1\ \{X > Y\}$
- $\{Y = 1\}\ X := Y\ \{X = 1 \wedge Y = 1\}$

Rule of sequential composition

$$\frac{\{\ P\ \}\ C_1\ \{\ R\ \} \qquad \{\ R\ \}\ C_2\ \{\ Q\ \}}{\{\ P\ \}\ C_1;\ C_2\ \{\ Q\ \}}$$

To prove that

If P holds before executing C_1; C_2,

then Q holds after executing C_1; C_2

one must find an assertion R such that

$\{\ P\ \}\ C_1\ \{\ R\ \}$ and $\{\ R\ \}\ C_2\ \{\ Q\ \}$

Example

The goal

$\{ Y = N \}$ X := Y+1; Y := X/2 $\{ X = N+1 \wedge Y = (N+1)/2 \}$

can be decomposed into subgoals

$$\{ Y = N \} \; X := Y+1 \; \{ X = N+1 \wedge Y = N \}$$

and

$$\{ X = N+1 \wedge Y = N \} \; Y := X/2 \; \{ X = N+1 \wedge Y = (N+1)/2 \}$$

Exercise: Prove the subgoals

Exercise

Prove the following goals:

- First step of factorial function:

  $\{ N \geq 0 \}$ X := N; R := 1 $\{ N \geq 0 \wedge X = N \wedge R = 1 \}$

- $\{ X \geq 0 \}$ X := X+1; Y := 1 $\{ X > Y \}$
- Swap using a temporary variable:

  $\{ X = X_0 \wedge Y = Y_0 \}$ T := X; X := Y; Y := T $\{ X = Y_0 \wedge Y = X_0 \}$

- Swap without temporary variable:

  $\{ X = X_0 \wedge Y = Y_0 \}$ X := X-Y; Y := X+Y; X := Y-X $\{ X = Y_0 \wedge Y = X_0 \}$

Hint about sequential composition

Consider a proof goal of the form

$$\{ P \}\ C;\ X := E\ \{ Q \}$$

a good strategy is to decompose this goal into the subgoals

$$\{ P \}\ C\ \{ Q[X := E] \}$$

and

$$\{ Q[X := E] \}\ X := E\ \{ Q \}$$

(the second subgoal holds trivially)

$Q[X := E]$ is the weakest precondition that ensures Q holds after $X := E$

Rule of if-then-else conditional

$$\frac{\{\, P \wedge E \,\}\ C_1\ \{\, Q \,\} \qquad \{\, P \wedge \neg E \,\}\ C_2\ \{\, Q \,\}}{\{\, P \,\}\ \textbf{if}\ E\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{end}\ \{\, Q \,\}}$$

- Precondition P can be strenghtened in the subgoals, depending whether condition E holds or not:
  - C_1 is executed only if E holds: $P \wedge E$
  - C_2 is executed only if $\neg E$ holds: $P \wedge \neg E$

Example

$\{ \text{ true } \}$ **if** $X < 0$ **then** $X := -X$ **else skip end** $\{ X \geq 0 \}$

can be decomposed into

$\{ X < 0 \}\ X := -X\ \{ X \geq 0 \}$

and

$\{ \neg(X < 0) \}$ **skip** $\{ X \geq 0 \}$

Exercise: prove the subgoals

Exercise

Prove formally the following goals:

- { true } **if** X = 1 **then** Y := 0 **else** Y := 1 **end** { Y < 2 }
- { true }

  if X mod 2 ≠ 0 **then** X := X+1 **else skip end**

  { X mod 2 = 0 }
- Maximum of two numbers:

  { true }

  if X > Y **then** M := X **else** M := Y **end**

  { $(M = X \lor M = Y) \land M \geq X \land M >= Y$ }

Rule of while loop

$$\frac{P \Rightarrow I \qquad \{ I \wedge E \}\ C_0\ \{ I \} \qquad I \wedge \neg E \Rightarrow Q}{\{ P \}\ \textbf{while}\ E\ \textbf{do}\ C_0\ \textbf{end}\ \{ Q \}}$$

This rule requires finding a property I, called a loop invariant, such that

- Initialisation: I holds before entering the loop: $P \Rightarrow I$
- Invariance: If I holds before executing the loop body C_0 then it still holds after: $\{ I \wedge E \}\ C_0\ \{ I \}$
- Consequence: Q can be deduced from I and the fact that the loop is terminated: $I \wedge \neg E \Rightarrow Q$

Example

- Factorial (on the black board)

  $\{N \geq 0 \wedge X = N \wedge R = 1\}$

  while $X \neq 0$ **do** $R := R * X; X := X - 1$ **end**

  $\{R = N!\}$

- Invariant: $(X \geq 0) \wedge (R = N!/X!)$

Exercise

Propose an invariant and prove

$\{ N \geq 0 \wedge X = 0 \wedge R = 1 \}$

```
while X < N do
        R := 2*R;
        X := X+1
end
```

$\{ R = 2^N \}$

Exercise

Prove the following goal (Euclidian division):

$\{ N \geq 0 \land M > 0 \}$

```
Q := 0;
R := N;
while R ≥ M do
    R := R-M;
    Q := Q+1
end
```

$\{ N = Q \times M + R \land M > R \land R \geq 0 \land Q \geq 0 \}$

Hint: use the loop invariant $N = Q \times M + R \land R \geq 0 \land Q \geq 0$

Finding the Right Invariant

- Finding the right loop invariant may be hard:
  - If property is too strong then invariance cannot be proven
  - If property is too weak then consequence cannot be proven
- But not harder than writing a correct program!
- Thinking in terms of preconditions, postconditions and loop invariants reduces the risk of implementation errors

Note about Program Termination

- For a program to be totally correct, one must also prove that it terminates
- The presented rules allow to prove almost anything on non-terminating programs
  Example:

  {X=1} **while** true **do skip end** {X=2}

  because $X = 1 \land \neg true \Rightarrow X = 2$ holds

Rule taking termination into account

$$\frac{P \Rightarrow I \quad \{ I \wedge E \wedge t = z \}\ C_0\ \{ I \wedge t < z \} \quad I \Rightarrow t \geq 0 \quad I \wedge \neg E \Rightarrow Q}{\{ P \}\ \textbf{while}\ E\ \textbf{do}\ C_0\ \textbf{end}\ \{ Q \}}$$

where z is a variable that is not affected by the loop body and t is an integer expression such that:

- t decreases at each loop iteration: $\{ I \wedge E \wedge \underline{t = z} \}\ C_0\ \{ I \wedge \underline{t < z} \}$
- t is always positive: $I \Rightarrow t \geq 0$

thus t cannot decrease infinitely and the loop terminates

t is called the loop variant

Exercise

Prove that

- Factorial (slide 42) is totally correct
  Hint: use X as loop variant
- Power of 2 (slide 43) is totally correct
  Hint: use N-X as loop variant
- Euclidian division (slide 44) is totally correct
  Hint: use R as loop variant
- What is the problem if in the precondition of Euclidian division $M > 0$ is weakened into $M \geq 0$?

Programming with assertions: design by contract

- A methodology proposed by B. Meyer (1986) and first implemented in the Eiffel language
- Write contract (*what should be done*) together with code (*how this is done*):
  - A pre and a postcondition with each function
  - An invariant and a variant with each loop
  - In OO programming, a property on state variables that should hold before and after every method call, named class invariant
- Contracts can be checked at runtime or, if the programming and assertion languages have formal semantics, connexion to provers is possible

Other features of design by contract

- Application to class inheritance (redefinition, dynamic binding)
- Application to exception handling
- Connection with automatic software documentation

Read more on contracts:

B. Meyer. *Object-Oriented Software Construction* (2nd edition). Prentice Hall. 1997.

Tools Related to Hoare Logic (1/4)

Educational tools

- **HAHA** (Univ. Warsaw, Poland)
  http://haha.mimuw.edu.pl
- **Java+ITP** (Univ. Illinois, USA)
  http://maude.cs.uiuc.edu/tools/javaitp
- **KeY** (Karlsruhe IT & TU Darmstadt, Germany, and Chalmers Univ., Sweden)
  http://www.key-project.org

Tools Related to Hoare Logic (2/4)

- **Frama-C** (CEA and Inria, France) http://www.frama-c.com
  - A modular environment for verifying (a subset of) C
  - Jessie plugin implements Hoare logic: automatic extraction of properties from annotated programs
  - Connection to external first-order logic provers

Tools Related to Hoare Logic (3/4)

B method (ClearSy, France)
http://www.methode-b.com/en

- Generalization of Hoare logic to nondeterministic specification language
- Requires systematic definition of preconditions, postconditions, and invariants
- First-order logic properties (proof obligations) are extracted automatically in order to be proven (using automatic or interactive provers)
- Includes a refinement method from abstract specification to executable program
- Used in several industrial projects (railway), several tools available

Tools Related to Hoare Logic (4/4)

SPEC# (Microsoft Research)
http://research.microsoft.com/en-us/projects/specsharp

- Formal language for contracts
- Extends C#, integrated in Visual Studio
- Boogie program verifier connected to automatic prover of logic properties
- Web interface (for toy examples): http://rise4fun.com/SpecSharp
- **Homework**: watch http://channel9.msdn.com/Blogs/Peli/The-Verification-Corner-Specifications-in-Action-with-SpecSharp

Factorial in SPEC#

```
class Factorial {
    static int fact (int n)
    requires n >= 0;
    ensures
      result == product{int i in (1:n+1); i};
      /* product of ints from 1 to n */
    {
        int x, r;
        x = n;
        r = 1;
    }
```

```
while (x > 0)
        invariant x >= 0;
        invariant x <= n;
        invariant
          r == product{int i in (x+1:n+1); i};
          /* product of ints from x+1 to n */
        {
            r *= x;
            x--;
        }
        return r;
    }
```

Conclusion

Beyond formal proof, assertions radically change the nature of software development in several ways:

- **Design aid**: build program + arguments that justify its correctness
- **Testing and debugging**: assertions can be checked at runtime
- **Documentation**: non-ambiguous and concise description of what the program does (instead of how this is done)

Competence and Knowledge which will be evaluated

- be able to
  - write simple **program properties**
  - guess **simple assertions**
  - **reason rigorously** on simple programs
- know
  - the notions of **precondition**, **postcondition**, **variant** and **invariant**
  - the proof **philosophy**

Proof of factorial

Define: $P = (X = N) \wedge (N \geq 0) \wedge (R = 1)$ (precondition)

$I(X, R) = (X \geq 0) \wedge (R = N!/X!)$ (loop invariant)

$(X \geq 0) \wedge (R = N!/X!) \wedge (X \neq 0) \Rightarrow (X - 1 \geq 0) \wedge (R*X = N!/(X-1)!)$

$$\dfrac{\dfrac{I(X,R) \wedge (X \neq 0) \Rightarrow I(X-1, R*X)}{\{ I(X, R) \wedge (X \neq 0) \}\ R := R*X\ \{ I(X-1, R) \}} \quad \dfrac{I(X-1, R) \Rightarrow I(X-1, R)}{\{ I(X-1, R)\}\ X := X-1\ \{ I(X, R) \}}}{P \Rightarrow I(X, R) \quad \{ I(X, R) \wedge (X \neq 0) \}\ R := R*X;\ X := X-1\ \{ I(X,R) \} \quad I(X,R) \wedge X = 0 \Rightarrow R = N!}$$

$$\overline{\{ P \}\ \textbf{while}\ X \neq 0\ \textbf{do}\ R := R*X;\ X := X-1\ \textbf{end}\ \{ R = N! \}}$$

More exercises

Which of the following goals are true?

- { true } X := 5 { X = 5 }
- { X = 2 } X := X + 1 { X = 3 }
- { true } X := X + 1 { X = X + 1 }
- { true } **skip** { false }
- { false } **skip** { true }
- { X = 0 } **while** X = 0 **do** X := X + 1 **end** { X = 1 }

More exercises for skilled students

- Is { P } C { true } always provable? Give an intuition. How would you prove this formally? *Hint*: use *induction* on programs
- Is { false } C { Q } always provable? Give an intuition. How would you prove this formally?
- Propose a formal deduction rule for

  { P } **do** C **until** E **end** { Q }