Lecture 11 – Systematic Program Proofs

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- Axiomatic Semantics
  - Hoare Triples
  - Partial Correctness vs. Total Correctness
  - Assertion Language
  - Denotational Semantics of Assertions
  - Satisfaction
  - Validity
- Hoare Logic
  - Example – Factorial
  - Soundness and Completeness
  - Relative Completeness

Contents

Contents

Motivation

Hoare logic is **non-deterministic** in its application, as it requires 'guessing' appropriate intermediate assertions for sequential composition and finding suitable assertions for the rule of consequence.

$$\textsc{Seq}\ \frac{\vdash \{\phi\}\ s_1\ \{\phi''\} \qquad \vdash \{\phi''\}\ s_2\ \{\phi'\}}{\vdash \{\phi\}\ s_1 ; s_2\ \{\phi'\}}$$

$$\textsc{Consequence}\ \frac{\vdash \phi \Rightarrow \phi_1 \qquad \vdash \{\phi_1\}\ s\ \{\phi_1'\} \qquad \vdash \phi_1' \Rightarrow \phi'}{\vdash \{\phi\}\ s\ \{\phi'\}}$$

To resolve this non-determinism, we can use **the weakest liberal precondition**, which allows us to calculate the necessary assertions mechanically by reasoning backward from the postcondition.

Weakest Liberal Precondition

Intuitively, the weakest liberal precondition of a statement s with respect to a postcondition ϕ is the weakest assertion that guarantees that ϕ holds after executing s.

Definition (Weakest Liberal Precondition)

$\mathbf{wlp}(s, \phi)$ is the **weakest liberal precondition** of statement s with respect to postcondition ϕ if:

$$\forall \sigma, I.\ \sigma \models_I \mathbf{wlp}(s, \phi) \iff S[\![s]\!](\sigma) \models_I \phi$$

Assertions $x < 3$ or $x < 5$ would both be sufficient preconditions for x := x + 1 to ensure that $x < 10$ holds afterwards.

$$\mathbf{wlp}(x := x + 1, x < 10) = x < 9$$

However, the weakest liberal precondition is the least restrictive assertion that still guarantees the postcondition. In this case, it would be $x < 9$.

Weakest Liberal Precondition

PLRG

$$
\begin{array}{lcl}
\mathbf{wlp}(\texttt{skip}, \phi) & = & \phi \\
\mathbf{wlp}(x \texttt{ := } a, \phi) & = & \phi[x \mapsto a] \\
\mathbf{wlp}(s_1; s_2, \phi) & = & \mathbf{wlp}(s_1, \mathbf{wlp}(s_2, \phi)) \\
\mathbf{wlp}(\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2, \phi) & = & (e \Rightarrow \mathbf{wlp}(s_1, \phi)) \wedge \\
 & & (\neg e \Rightarrow \mathbf{wlp}(s_2, \phi)) \\
\mathbf{wlp}(\texttt{while } e \texttt{ do } s, \phi) & = & \mathbf{gfp} F = \bigwedge_{i \geq 0} F^i(\texttt{true})
\end{array}
$$

where

$$F(\phi') = (e \Rightarrow \mathbf{wlp}(s, \phi')) \wedge (\neg e \Rightarrow \phi)$$

However, it is impossible to compute $\mathbf{wlp}$ for loops in general, as it may require an infinite number of iterations to reach a fixed point.

Weakest Liberal Precondition – Loop Invariant

To overcome the challenge of computing $\mathbf{wlp}$ for loops, we can introduce the concept of a **loop invariant**, which is an assertion that holds before and after each iteration of the loop.

Assume that a **loop invariant** $\phi_{\mathbb{I}}$ is given for $\texttt{while}\ e\ \texttt{do}\ s$:

$$\texttt{while}\ e\ @[\phi_{\mathbb{I}}]\ \texttt{do}\ s$$

Then, we can modify the definition of $\mathbf{wlp}$ for loops as follows:

$$\begin{aligned}&\mathbf{wlp}(\texttt{while}\ e\ @[\phi_{\mathbb{I}}]\ \texttt{do}\ s, \phi)\\ =\ &\underbrace{\phi_{\mathbb{I}}}_{\text{1. initial}} \wedge \underbrace{(\forall \overline{x'}.\ (e \wedge \phi_{\mathbb{I}} \Rightarrow \mathbf{wlp}(s, \phi_{\mathbb{I}}))[\overline{x \mapsto x'}])}_{\text{2. preservation}} \wedge \underbrace{(\forall \overline{x}.\ (\neg e \wedge \phi_{\mathbb{I}} \Rightarrow \phi)[\overline{x \mapsto x'}])}_{\text{3. termination}}\end{aligned}$$

where $\overline{x}$ are the **modified variables** in s and $\overline{x'}$ are **fresh variables**.

Example – Factorial

PLRG

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {

    x := x * i;

    i := i - 1;

}
```

$\{x = n!\}$

Example – Factorial

PLRG

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {

    x := x * i;

    i := i - 1;
    {0 ≤ i' ∧ x' × i'! = n!}
}
{x = n!}
```

Example – Factorial

PLRG

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {
    x := x * i;
```
$\{0 < i' \wedge x' \times (i' - 1)! = n!\}$
```
    i := i - 1;
```
$\{0 \leq i' \wedge x' \times i'! = n!\}$
```
}
```
$\{x = n!\}$

Example – Factorial

PLRG

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {
    {0 < i' ∧ x' × i'! = n!}
    x := x * i;
    {0 < i' ∧ x' × (i' − 1)! = n!}
    i := i - 1;
    {0 ≤ i' ∧ x' × i'! = n!}
}
{x = n!}
```

Example – Factorial

PLRG

1. initial: $\phi_{\mathbb{I}}$

2. preservation: $\overbrace{(\forall \overline{x'}.\ (e \wedge \phi_{\mathbb{I}} \Rightarrow \mathbf{wlp}(s, \phi_{\mathbb{I}}))[\overline{x \mapsto x'}])}$

$\wedge \ldots$

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {
```

$\{0 < i' \wedge x' \times i'! = n!\}$

```
    x := x * i;
```

$\{0 < i' \wedge x' \times (i' - 1)! = n!\}$

```
    i := i - 1;
```

$\{0 \leq i' \wedge x' \times i'! = n!\}$

```
}
```

$\{x = n!\}$

Example – Factorial

PLRG

1. initial: $\phi_{\mathbb{I}}$

3. termination: $\wedge\, (\forall \overline{x}.\ (\neg e \wedge \phi_{\mathbb{I}} \Rightarrow \phi)[\overline{x \mapsto x'}])$

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {
```

$\{0 < i' \wedge x' \times i'! = n!\}$

```
    x := x * i;
```

$\{0 < i' \wedge x' \times (i' - 1)! = n!\}$

```
    i := i - 1;
```

$\{0 \leq i' \wedge x' \times i'! = n!\}$

```
}
```

$\{x = n!\}$

Example – Factorial

$0 \leq i \wedge x \times i! = n!$

```
while (0 < i) @[0 ≤ i ∧ x × i! = n!] {
```

$\{0 < i' \wedge x' \times i'! = n!\}$

```
    x := x * i;
```

$\{0 < i' \wedge x' \times (i' - 1)! = n!\}$

```
    i := i - 1;
```

$\{0 \leq i' \wedge x' \times i'! = n!\}$

```
}
```

$\{x = n!\}$

The precondition implies the $\mathbf{wlp}$ of the statement with postcondition:

$$0 \leq n \wedge i = n \wedge x = 1 \quad \Longrightarrow \quad 0 \leq i \wedge x \times i! = n!$$

Thus, we can conclude that the Hoare triple is valid:

$$\vdash \{0 \leq n \wedge i = n \wedge x = 1\}\ s\ \{x = n!\}$$

Provability and Correctness of $\mathbf{wlp}$

PLRG

Theorem (Provability of $\mathbf{wlp}$)

For any statement s and assertion ϕ, the following holds:

$$\vdash \{\mathbf{wlp}(s, \phi)\}\ s\ \{\phi\}$$

The Hoare triple formed by $\mathbf{wlp}$ is not only semantically valid but also formally derivable ($\vdash$) using the deductive rules of Hoare logic.

Theorem (Correctness of $\mathbf{wlp}$)

For any statement s and assertions ϕ and ϕ', the following holds:

$$\models \{\phi\}\ s\ \{\phi'\} \quad \Longrightarrow \quad \models \phi \Rightarrow \mathbf{wlp}(s, \phi')$$

Since $\mathbf{wlp}$ is the weakest assertion that guarantees the postcondition, any valid Hoare triple with the same statement and postcondition must have a precondition that implies $\mathbf{wlp}$.

Relative Completeness of Hoare Logic

Recall the definition of relative completeness:

Theorem (Relative Completeness)

If we assume that all valid assertions are provable in the assertion logic, then all valid partial correctness statements are derivable in Hoare logic:

$$(\forall \phi.\ \models \phi \implies \vdash \phi) \implies (\forall \phi, s, \phi'.\ \models \{\phi\}\ s\ \{\phi'\} \implies \vdash \{\phi\}\ s\ \{\phi'\})$$

We can prove this by using the $\mathbf{wlp}$.

- By the correctness of $\mathbf{wlp}$, we have: $\models \phi \Rightarrow \mathbf{wlp}(s, \phi')$.
- By the assumption of the oracle (all valid assertions are provable), we can derive: $\vdash \phi \Rightarrow \mathbf{wlp}(s, \phi')$.
- By the provability of $\mathbf{wlp}$, we have: $\vdash \{\mathbf{wlp}(s, \phi')\}\ s\ \{\phi'\}$.
- By the consequence rule, we can derive: $\vdash \{\phi\}\ s\ \{\phi'\}$:

$$\textsc{Consequence}\ \frac{\vdash \phi \Rightarrow \mathbf{wlp}(s, \phi') \qquad \vdash \{\mathbf{wlp}(s, \phi')\}\ s\ \{\phi'\} \qquad \vdash \phi' \Rightarrow \phi'}{\vdash \{\phi\}\ s\ \{\phi'\}}$$

Contents

Dafny

- Dafny is a programming language and verification tool that allows developers to write code and specify its behavior using preconditions, postconditions, and invariants.
- It uses a combination of static verification and runtime checks to ensure that the code adheres to its specifications.
- Dafny supports features such as classes, methods, functions, and inductive datatypes, making it suitable for a wide range of applications.

Example – Factorial

```
method Factorial(n: nat) returns (x: nat)
  requires 0 <= n                 // precondition
  ensures x == factorial(n)       // postcondition
{
  var i := n;
  x := 1;
  while 0 < i
    invariant 0 <= i              // invariants
    invariant x * factorial(i) == factorial(n)
  {
    x := x * i;
    i := i - 1;
  }
  return x;
}
function factorial(n: nat): nat {
  if n == 0 then 1 else n * factorial(n - 1)
}
```

Example – Factorial (Wrong 1)

PLRG

```
method Factorial(n: nat) returns (x: nat)
  requires 0 <= n                // precondition
  ensures x == factorial(n)      // postcondition
{
  var i := n;
  x := 1;
  while 0 < i
    invariant n < i // violated on entry
    invariant x * factorial(i) == factorial(n)
  {
    x := x * i;
    i := i - 1;
  }
  return x;
}
function factorial(n: nat): nat {
  if n == 0 then 1 else n * factorial(n - 1)
}
```

Example – Factorial (Wrong 2)

PLRG

```
method Factorial(n: nat) returns (x: nat)
  requires 0 <= n                // precondition
  ensures x == factorial(n)      // postcondition
{
  var i := n;
  x := 1;
  while 0 < i
    invariant n <= i // invariant does not preserve
    invariant x * factorial(i) == factorial(n)
  {
    x := x * i;
    i := i - 1;
  }
  return x;
}
function factorial(n: nat): nat {
  if n == 0 then 1 else n * factorial(n - 1)
}
```

Example – Factorial (Wrong 3)

```
method Factorial(n: nat) returns (x: nat)
  requires 0 <= n                 // precondition
  ensures x == factorial(n)       // postcondition
{
  var i := n;
  x := 1;
  while 0 < i
    invariant 0 <= i
    invariant x * factorial(i) <= factorial(n) // violate postcondition
  {
    x := x * i;
    i := i - 1;
  }
  return x;
}
function factorial(n: nat): nat {
  if n == 0 then 1 else n * factorial(n - 1)
}
```

Example – Fibonacci

```
method ComputeFib(n: nat) returns (x: nat)
  requires 0 <= n                 // precondition
  ensures x == fib(n)             // postcondition
{
  if n == 0 { return 0; }
  var i, prev, cur := 1, 0, 1;
  while i < n
    invariant i <= n              // invariants
    invariant prev == fib(i - 1)
    invariant cur == fib(i)
  {
    prev, cur := cur, prev + cur;
    i := i + 1;
  }
  return cur;
}
function fib(n: nat): nat {
  if n == 0 then 0 else if n == 1 then 1 else fib(n - 1) + fib(n - 2)
}
```

Example – IndexOf

```
method IndexOf(a: array<int>, v: int) returns (index: int)
  ensures -1 <= index < a.Length
  ensures index != -1 ==> a[index] == v
  ensures index != -1 ==> forall k :: 0 <= k < index ==> a[k] != v
  ensures index == -1 ==> forall k :: 0 <= k < a.Length ==> a[k] != v
{
  var i := 0;
  while i < a.Length
    invariant 0 <= i <= a.Length
    invariant forall k :: 0 <= k < i ==> a[k] != v
  {
    if a[i] == v { return i; }
    i := i + 1;
  }
  return -1;
}
```

Next Lecture

- Separation Logic (1)

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr