Theme 2: Proving Correct Imperative Sequential Programs

Lectures 4 & 5:
Partial Correctness of Imperative Programs – Hoare Logic

Ahmed Bouajjani

Paris Diderot University, Paris 7

January 2014
Imperative Sequential Programs

- Let $X$ be a set of typed variables declared in the program.
- Values of variables range over a data domain $D$. Let $Op$ be a set of operations and let $Rel$ be a set of relations over $D$.
- The statements in a program are defined as follows:

$$S ::= \text{skip} \mid x := E \mid S ; S \mid \text{if } C \text{ then } S \text{ else } S \mid \text{while } C \text{ do } S$$

where $E$ is a term and $C$ is a formula over $X$ in $\text{FO}(D, Op, Rel)$. 
Example of a program

\[ f : \text{Nat} ; \]

\[
\text{ifact}(n : \text{Nat}) =
\]

\[
i : \text{Nat} ;
\]

\[
f := 1 ;
\]

\[
i := 0 ;
\]

\[
\text{while } i \neq n \text{ do}
\]

\[
i := i + 1 ;
\]

\[
f := i \ast f
\]
Another example of a program

\[
\text{r : Nat ;}
\]
\[
\text{isum (}\ell : \text{List[Nat]) =}
\]
\[
\ell' : \text{List[Nat] ;}
\]
\[
r := 0 ;
\]
\[
\ell' := \ell ;
\]
\[
\text{while } \ell' \neq [] \text{ do}
\]
\[
r := r + \text{head}(\ell') ;
\]
\[
\ell' := \text{tail}(\ell')
\]
Program semantics

- Imperative programs transform memory states.
- A program is seen as a state machine.
- A state corresponds to a valuation of the program variables:
  \[ \mu : X \rightarrow D \]
- Transitions between states correspond to the execution of statements:
  \[ \mu \xrightarrow{S} \mu' \]
Semantics: Transition rules

\[ \mu \xrightarrow{\text{skip}} \mu \]

\[ \mu \leftarrow x := \exp \rightarrow \mu[x \leftarrow d] \]

\[ \mu \xrightarrow{S_1} \nu \quad \nu \xrightarrow{S_2} \mu' \]

\[ \mu \xrightarrow{S_1;S_2} \mu' \]

\[ \mu \models C \quad \mu \xrightarrow{S_1} \mu' \]

\[ \mu \xrightarrow{\text{if } C \text{ then } S_1 \text{ else } S_2} \mu' \]

\[ \mu \models \neg C \quad \mu \xrightarrow{S_2} \mu' \]

\[ \mu \xrightarrow{\text{while } C \text{ do } S} \mu \]

\[ \mu \xrightarrow{\text{while } C \text{ do } S} \mu' \]

\[ \langle \exp \rangle \mu = d \]
Assertions

- Assertions about program states can be expressed in FO logic over X.
- We consider two special statements: `assume(\phi)` and `assert(\phi)` where \(\phi\) is a FO formula over X.

```plaintext
f : Nat;
ifact (n : Nat) =
    assume(true);
i : Nat;
f := 1;
i := 0;
while i \neq n do
    i := i + 1;
f := i \times f;
assert(f = fact(n))
```
Assertions

- Assertions about program states can be expressed in FO logic over X.

- We consider two special statements: $\text{assume}(\phi)$ and $\text{assert}(\phi)$ where $\phi$ is a FO formula over X.

```plaintext
r : Nat ;

isum (\ell : List[Nat]) =
   assume(true);
\ell' : List[Nat] ;
r := 0 ;
\ell' := \ell ;
while \ell' \neq [] do
   r := r + head(\ell') ;
   \ell' := tail(\ell') ;
assert(r = \Sigma(\ell))
```
Assertions

- Assertions about program states can be expressed in FO logic over X.

- We consider two special statements: `assume(\phi)` and `assert(\phi)` where \( \phi \) is a FO formula over X.

```plaintext
r : Nat ;

isum (\ell : List[Nat]) =
  assume(\forall e \in \star. \text{ln}(e, \ell) \Rightarrow (e = 1))

\ell' : List[Nat] ;

x := 0 ;

\ell' := \ell ;

while \ell' \neq [] do
  r := r + head(\ell') ;
  \ell' := tail(\ell') ;

assert(r = |\ell|)
```
Assume – Assert statements: Semantics

- Let \( \bot \) be a special error state

- Transition rules:

\[
\mu \models \phi \\
\frac{}{\mu \xrightarrow{\text{assume}(\phi)} \mu}
\]

\[
\mu \models \phi \\
\frac{}{\mu \xrightarrow{\text{assert}(\phi)} \mu}
\]

\[
\mu \models \neg \phi \\
\frac{}{\mu \xrightarrow{\text{assert}(\phi)} \bot}
\]
Loop Invariants

\[ f : \text{Nat} ; \]

\[ \text{ifact} \ (n : \text{Nat}) = \]

\[ \text{assume}(\text{true}) ; \]

\[ i : \text{Nat} ; \]

\[ f := 1 ; \]

\[ i := 0 ; \]

\[ \text{while } i \neq n \text{ do} \]

\[ \text{invariant}(?) ; \]

\[ i := i + 1 ; \]

\[ f := i * f ; \]

\[ \text{assert}(f = \text{fact}(n)) \]

- A property that is true initially, and after each iteration.
Loop Invariants

\[ f : \text{Nat} ; \]
\[ \text{ifact} (n : \text{Nat}) = \]
\[ \text{assume}(true); \]
\[ i : \text{Nat} ; \]
\[ f := 1 ; \]
\[ i := 0 ; \]
\[ \text{while } i \neq n \text{ do} \]
\[ \text{invariant}(?); \]
\[ i := i + 1 ; \]
\[ f := i \times f ; \]
\[ \text{assert}(f = \text{fact}(n)) \]

- A property that is true initially, and after each iteration.
- But there are many invariants!!: true, \(i \geq 0\), \(f \geq 1\), ...
Loop Invariants

\[
f : \text{Nat} ;
\]

\[
\text{ifact} (n : \text{Nat}) =
\]
\[
\text{assume}(\text{true}) ;
\]

\[
i : \text{Nat} ;
\]

\[
f := 1 ;
\]

\[
i := 0 ;
\]

\[
\text{while } i \neq n \text{ do}
\]
\[
\text{invariant}(?) ;
\]

\[
i := i + 1 ;
\]

\[
f := i \ast f ;
\]

\[
\text{assert}(f = \text{fact}(n))
\]

- A property that is true initially, and after each iteration.
- But there are many invariants!!: true, \( i \geq 0 \), \( f \geq 1 \), ...

- A “useful invariant”:
  
  After the last iteration, it implies the desired post-condition.
Loop Invariants

\[ f : \text{Nat} ; \]
\[ \text{ifact (n : Nat) =} \]
\[ \quad \text{assume(true);} \]
\[ i : \text{Nat} ; \]
\[ f := 1 ; \]
\[ i := 0 ; \]
\[ \text{while } i \neq n \text{ do} \]
\[ \quad \text{invariant}(f = \text{fact}(i)); \]
\[ \quad i := i + 1 ; \]
\[ \quad f := i \ast f ; \]
\[ \text{assert}(f = \text{fact}(n)) \]

- A property that is true initially, and after each iteration.
- But there are many invariants!!: \text{true}, \ f \geq 1, \ ...
- A “useful invariant”:
  \text{After the last iteration, it implies the desired post-condition.}
Programming methodology

- Define the states of the programs (variables and their types).
- Define the (assumed) initial and the (ensured) last state.
- Define iterative computations: Provide loop invariants.
Example: Reversing a list

\[ \rho : \text{List}[\star] ; \]
\[ \text{irev } (\ell : \text{List}[\star]) = \]
\[ \quad \text{assume(true)}; \]
\[ \]
\[ \text{assert}(\rho = \text{Rev}(\ell)) \]
Example: Reversing a list

\[ \rho : \text{List}[\ast] ; \]

\[ \text{irev} \ (\ell : \text{List}[\ast]) = \]
\[ \begin{align*}
\ &\text{assume(true);} \\
\ &\ell' : \text{List}[\ast] ; \\
\ &\rho := []; \% \rho \ is \ the \ reverse \ of \ the \ treated \ prefix \ of \ \ell \\
\ &\ell' := \ell ; \% \ell' \ is \ the \ non-treated \ suffix \ of \ \ell \\
\ &\text{while} \quad \text{do} \quad \\
\end{align*} \]

\[ \text{assert(} \rho = \text{Rev}(\ell) \text{)} \]
Example: Reversing a list

\[\rho : \text{List}[\ast] ;\]

\[\text{irev } (\ell : \text{List}[\ast]) =\]
\[\text{assume(true);}\]
\[\ell' : \text{List}[\ast] ;\]
\[\rho : = [] ; \quad \% \rho \text{ is the reverse of the treated prefix of } \ell\]
\[\ell' : = \ell ; \quad \% \ell' \text{ is the non-treated suffix of } \ell\]
\[\text{while } \text{do }\]
\[\text{invariant(} \ell = \text{Rev}(\rho)@\ell'\text{)}\]

\[\text{assert}(\rho = \text{Rev}(\ell))\]
Example: Reversing a list

\[ \rho : \text{List}\[\star]\; ; \]
\[ \text{irev}(\ell : \text{List}\[\star]\)) = \]
\[ \text{assume(true)}; \]
\[ \ell' : \text{List}\[\star]\; ; \]
\[ \rho := []; \quad \% \rho \text{ is the reverse of the treated prefix of } \ell \]
\[ \ell' := \ell; \quad \% \ell' \text{ is the non-treated suffix of } \ell \]
\[ \text{while } \ell' \neq [] \text{ do} \]
\[ \quad \text{invariant}(\ell = \text{Rev}(\rho)@\ell') \]
\[ \quad \rho := \text{head}(\ell') \cdot \rho ; \]
\[ \quad \ell' := \text{tail}(\ell') ; \]
\[ \quad \text{assert}(\rho = \text{Rev}(\ell)) \]
Pre-post condition reasoning

- Consider formulas of the form:

\[
\{ \phi \} \ S \ \{ \psi \}
\]

where \( S \) is a statement, and \( \phi \) and \( \psi \) are assertions.

- \( \phi \) is the pre-condition, and \( \psi \) is the post-condition.
Pre-post condition reasoning

- Consider formulas of the form:

\[ \{ \phi \} \; S \; \{ \psi \} \]

where \( S \) is a statement, and \( \phi \) and \( \psi \) are assertions.

- \( \phi \) is the pre-condition, and \( \psi \) is the post-condition.

- Formal Semantics:

\[ \{ \phi \} \; S \; \{ \psi \} \; \text{iff} \; \forall \mu, \mu'. (\mu \models \phi \land \mu \xrightarrow{S} \mu') \Rightarrow \mu' \models \psi \]

- Intuitive meaning:

Starting from a state satisfying \( \phi \), if the execution of \( S \) terminates, then the reached state must satisfy \( \psi \).
Pre-post condition reasoning

- Consider formulas of the form:
  \[
  \{ \phi \} \text{ } S \{ \psi \}
  \]
  where \( S \) is a statement, and \( \phi \) and \( \psi \) are assertions.
- \( \phi \) is the pre-condition, and \( \psi \) is the post-condition.
- Formal Semantics:
  \[
  \{ \phi \} \text{ } S \{ \psi \} \text{ iff } \forall \mu, \mu'. (\mu \models \phi \land \mu \xrightarrow{S} \mu') \Rightarrow \mu' \models \psi
  \]
- Intuitive meaning:
  Starting from a state satisfying \( \phi \), if the execution of \( S \) terminates, then the reached state must satisfy \( \psi \).
- Problem: How to prove the validity of such formulas?
A Formal System: Hoare Logic

- A set of axioms and inference rules of the form:

\[
\begin{array}{c}
\text{Axiom} \\
\hline
\text{Premise}_1 \quad \cdots \quad \text{Premise}_N
\end{array}
\]

\[
\text{Conclusion}
\]
A Formal System: Hoare Logic

- A set of axioms and inference rules of the form:

  \[
  \begin{array}{c}
  \text{Axiom} \\
  \text{Premise}_1 & \cdots & \text{Premise}_N \\
  \hline
  \text{Conclusion}
  \end{array}
  \]

- Compositional reasoning using the structure of the programs:

  \[
  \begin{array}{c}
  \{\phi_1\} S_1 \{\psi_1\} & \cdots & \{\phi_N\} S_N \{\psi_N\} \\
  \{\phi\} \text{Comp}(S_1, \ldots, S_N) \{\psi\}
  \end{array}
  \]
Hoare Logic: Axioms for Basic Statements

\[
\{\phi\} \text{skip} \{\phi\}
\]
Hoare Logic: Axioms for Basic Statements

\[
\{ \phi \} \text{skip} \{ \phi \}
\]

\[
\{ \phi[\text{exp/}x] \} \ x := \text{exp} \{ \phi \}
\]
Hoare Logic: Axioms for Basic Statements

\[ \{\phi\} \text{ skip } \{\phi\} \]

\[ \{\phi[\text{exp}/x]\} x := \text{exp} \{\phi\} \]

?? \( x := x + 2 \) \( \{x \geq 5 \land x \leq y + 1\} \)
Hoare Logic: Axioms for Basic Statements

\[
\{\phi\} \text{skip} \{\phi\}
\]

\[
\{\phi[exp/x]\} x := \text{exp} \{\phi\}
\]

?? \(x := x + 2\) \(\{x \geq 5 \land x \leq y + 1\}\)

\(\{x + 2 \geq 5 \land x + 2 \leq y + 1\}\) \(x := x + 2\) \(\{x \geq 5 \land x \leq y + 1\}\)

\(\{x \geq 3 \land x + 1 \leq y\}\) \(x := x + 2\) \(\{x \geq 5 \land x \leq y + 1\}\)
Forward version of the assignment axiom?

- Let $M$ be a set of program states ($M \subseteq [X \rightarrow D]$), and let $S$ be a program statement.

- Sets of immediate successors and predecessors:
  
  $\text{post}(M, S) = \{ \mu' : \exists \mu \in M. \mu \xrightarrow{S} \mu' \}$
  
  $\text{pre}(M, S) = \{ \mu : \exists \mu' \in M. \mu \xrightarrow{S} \mu' \}$

- Let $\phi(X)$ be an assertion over $X$ such that $\left[ \phi \right] = M$. Assertions for $\text{post}(M, x := \text{exp}(X))$ and $\text{pre}(M, x := \text{exp}(X))$?
Forward version of the assignment axiom? (cont.)

- Assertions defining $\text{post}(M, x := \exp(X))$ and $\text{pre}(M, x := \exp(X))$:

  $\text{pre}(\phi, x := \exp)(X) = \exists X'. (\phi(X') \land X' = \exp(X))$

  $\text{post}(\phi, x := \exp)(X) = \exists X'. (\phi(X') \land X = \exp(X'))$

- The pre formula can be simplified (quantification elimination):

  $\phi_{\text{pre}}(X) = \phi[\exp(X)/X]$

- Can we do the same for the post formula?

  $\text{post}(2 \leq x \land x \leq y, x := y) = \exists x'. (2 \leq x' \land x' \leq y \land x = y) = 2 \leq y \land x = y$

- Quantification elimination depends on the data theory. Possible for, e.g., $\text{FO}(\mathbb{N}, \{0, 1, +\}, \{\leq\})$. Not always possible / expensive.
Hoare Logic: Sequential composition

\[
\begin{array}{c}
\{ \phi_1 \} \ S_1 \ {\phi_2} \quad \{ \phi_2 \} \ S_2 \ {\phi_3} \\
\{ \phi_1 \} \ S_1; \ S_2 \ {\phi_3}
\end{array}
\]
Example: Swap

\[
\begin{align*}
& t := x ; \\
& x := y ; \\
& y := t
\end{align*}
\]
Example: Swap

```
t := x ;
x := y ;
y := t
{x = a ∧ y = b}
```
Example: Swap

\[ t := x ; \]
\[ x := y ; \]
\[ \{ x = a \land b = t \} \]
\[ y := t \]
\[ \{ x = a \land y = b \} \]
Example: Swap

t := x ;
\{ y = a \land b = t \}

x := y ;
\{ x = a \land b = t \}

y := t
\{ x = a \land y = b \}
Example: Swap

\{y = a \land b = x\}
\quad t := x ;
\{y = a \land b = t\}
\quad x := y ;
\{x = a \land b = t\}
\quad y := t
\{x = a \land y = b\}
Hoare Logic: Implication rule

\[
\phi_1 \Rightarrow \phi'_1 \quad \{\phi'_1\} \text{ S } \{\phi'_2\} \quad \phi'_2 \Rightarrow \phi_2
\]

\[
\{\phi_1\} \text{ S } \{\phi_2\}
\]
Hoare Logic: Conditional rule

\[
\begin{align*}
\{\phi \land C\} & \quad S_1 \quad \{\phi'\} \\
\{\phi \land \neg C\} & \quad S_2 \quad \{\phi'\} \\
\hline
\{\phi\} & \quad \text{if } C \text{ then } S_1 \text{ else } S_2 \quad \{\phi'\}
\end{align*}
\]
Example: Minimum of 2 different values

- We want to establish:

\[
\begin{align*}
    \{\text{true}\} \\
    \text{if } x < y \text{ then } m := x \text{ else } m := y \\
    \{m \leq x \land m \leq y\}
\end{align*}
\]
Example: Minimum of 2 different values

- We want to establish:
  \[
  \{\text{true}\}
  \]
  if \( x < y \) then \( m := x \) else \( m := y \)
  \[
  \{m \leq x \land m \leq y\}
  \]

- Premises that must be proved:
  1. \( \{x < y\} \ m := x \ \{m \leq x \land m \leq y\}\)
  2. \( \{y < x\} \ m := y \ \{m \leq x \land m \leq y\}\)

Proof of Premise 1: Assignment axiom + implication rule

\[
□\ x < y \Rightarrow x \leq y
\]

Proof of Premise 2 is identical.
Example: Minimum of 2 different values

We want to establish:

\[ \{ \text{true} \} \]

\[ \text{if } x < y \text{ then } m := x \text{ else } m := y \]

\[ \{ m \leq x \land m \leq y \} \]

Premises that must be proved:

1. \( \{ x < y \} \ m := x \ \{ m \leq x \land m \leq y \} \)
2. \( \{ y < x \} \ m := y \ \{ m \leq x \land m \leq y \} \)

Proof of Premise 1: Assignment axiom + implication rule

- \( \{ x \leq x \land x \leq y \} \ m := x \ \{ m \leq x \land m \leq y \} \)
- \( x < y \Rightarrow x \leq y \)
Example: Minimum of 2 different values

- We want to establish:

\[
\begin{align*}
\{true\} \\
\text{if } x < y \text{ then } m := x \text{ else } m := y \\
\{m \leq x \land m \leq y\}
\end{align*}
\]

- Premises that must be proved:
  1. \(\{x < y\} \quad m := x \quad \{m \leq x \land m \leq y\}\)
  2. \(\{y < x\} \quad m := y \quad \{m \leq x \land m \leq y\}\)

- Proof of Premise 1: Assignment axiom + implication rule
  \(\text{▶ } \{x \leq x \land x \leq y\} \quad m := x \quad \{m \leq x \land m \leq y\}\)
  \(\text{▶ } x < y \Rightarrow x \leq y\)

- Proof of Premise 2 is identical.
Hoare Logic: Iteration rule

\[
\begin{align*}
\{\phi \land C\} \quad S \quad \{\phi\} \\
\{\phi\} \quad \text{while } C \text{ do } S \quad \{\phi \land \neg C\}
\end{align*}
\]
Example: Iterative factorial

- Assignment + Sequential composition rules:

\[
\begin{align*}
((i + 1) \cdot f = \text{fact}(i + 1)) \\
i &:= i + 1 \\
(i \cdot f = \text{fact}(i)) \\
f &:= i \cdot f ; \\
\{f = \text{fact}(i)\}
\end{align*}
\]
Example: Iterative factorial

- Assignment + Sequential composition rules:

\[
\{ (i + 1) \ast f = \text{fact}(i + 1) \} \\
i := i + 1 ; \\
\{ i \ast f = \text{fact}(i) \} \\
f := i \ast f ; \\
\{ f = \text{fact}(i) \}
\]

- Definition of \text{fact}: \text{fact}(i + 1) = (i + 1) \ast \text{fact}(i)
Example: Iterative factorial

- **Assignment + Sequential composition rules:**

  \[
  \{ (i + 1) \ast f = \text{fact}(i + 1) \} \\
  i := i + 1 ; \\
  \{ i \ast f = \text{fact}(i) \} \\
  f := i \ast f ; \\
  \{ f = \text{fact}(i) \}
  \]

- **Definition of \text{fact}:** \( \text{fact}(i + 1) = (i + 1) \ast \text{fact}(i) \)

- **Theory of integers:** \( f = \text{fact}(i) \) \( \implies \) \((i + 1) \ast f = (i + 1) \ast \text{fact}(i)\)
Example: Iterative factorial

- **Assignment + Sequential composition rules:**
  \[
  \{(i + 1) \ast f = \text{fact}(i + 1)\}
  \]
  \[
  i := i + 1 ;
  \]
  \[
  \{i \ast f = \text{fact}(i)\}
  \]
  \[
  f := i \ast f ;
  \]
  \[
  \{f = \text{fact}(i)\}
  \]

- **Definition of fact:** \(\text{fact}(i + 1) = (i + 1) \ast \text{fact}(i)\)

- **Theory of integers:** \(f = \text{fact}(i)) \implies (i + 1) \ast f = (i + 1) \ast \text{fact}(i)\)

- **Implication rule:**
  \[
  \{(f = \text{fact}(i))\}
  \]
  \[
  i := i + 1 ; f := i \ast f
  \]
  \[
  \{(f = \text{fact}(i))\}
  \]
Example: Iterative factorial (cont.)

So far:

\[
\begin{align*}
\{ f = \text{fact}(i) \} \\
i := i + 1 ; \ f := i \ast f \\
\{ f = \text{fact}(i) \}
\end{align*}
\]
Example: Iterative factorial (cont.)

- So far: + Implication rule

\[
\{ f = \text{fact}(i) \land i \neq n \} \\
i := i + 1 ; \ f := i \times f \\
\{ f = \text{fact}(i) \}
\]
So far: + Implication rule

\[ \{ f = \text{fact}(i) \land i \neq n \} \]

\[ i := i + 1; \ f := i \ast f \]

\[ \{ f = \text{fact}(i) \} \]

Iteration rule:

\[ \{ f = \text{fact}(i) \} \]

\[ \text{while} \ (i \neq n) \ \text{do} \ \{ i := i + 1; \ f := i \ast f \} \]

\[ \{ f = \text{fact}(i) \land i = n \} \]
Example: Iterative factorial (cont.)

- So far: + Implication rule
  \[
  \{ f = \text{fact}(i) \land i \neq n \} \\
  i := i + 1; \ f := i \ast f \\
  \{ f = \text{fact}(i) \}
  \]

- Iteration rule: + Implication rule
  \[
  \{ f = \text{fact}(i) \} \\
  \text{while } (i \neq n) \text{ do } \{ i := i + 1; \ f := i \ast f \} \\
  \{ f = \text{fact}(i) \land i = n \} \\
  \Rightarrow \\
  \{ f = \text{fact}(n) \}
  \]
Example: Iterative factorial (cont.)

\[
\text{ifact} (n : \text{Nat}) = \\
\text{assume(true);}
\]

\[
f := 1 ;
\]

\[
i := 0 ;
\]

\[
\text{while } i \neq n \text{ do}
\]

\[
i := i + 1 ;
\]

\[
f := i \ast f ;
\]

\[
\text{assert}(f = \text{fact}(n))
\]
Example: Iterative factorial (cont.)

\[
\text{ifact}(n : \text{Nat}) = \\
\begin{align*}
\text{assume}(true); \\
f &:= 1; \\
i &:= 0; \\
\text{while } i \neq n \text{ do} \quad \text{while } i \neq n \text{ do} \\
\quad i &:= i + 1; \\
\quad f &:= i \times f; \\
\quad \{ f = \text{fact}(i) \} \\
\quad \{ f = \text{fact}(n) \} \\
\text{assert}(f = \text{fact}(n))
\end{align*}
\]
Example: Iterative factorial (cont.)

\[ ifact(n : Nat) = \]
\[ assume(true); \]
\[ f := 1 ; \]
\[ i := 0 ; \]

while \( i \neq n \) do

\[ \{(i + 1) * f = fact(i + 1)\} \iff (i + 1) * f = (i + 1) * fact(i) \]
\[ i := i + 1 ; \]
\[ \{i * f = fact(i)\} \]
\[ f := i * f ; \]
\[ \{f = fact(i)\} \]
\[ \{f = fact(n)\} \]

\assert(f = fact(n))
Example: Iterative factorial (cont.)

\[
\text{ifact} \left( n : \text{Nat} \right) = \\
\text{assume(true);}
\]

\[
f := 1 ;
\]

\[
i := 0 ;
\]

\[
\text{while } i \neq n \text{ do}
\]
\[
\{ f = \text{fact}(i) \land i \neq n \} \implies
\{ (i + 1) \ast f = \text{fact}(i + 1) \} \iff (i + 1) \ast f = (i + 1) \ast \text{fact}(i)
\]
\[
i := i + 1 ;
\]
\[
\{ i \ast f = \text{fact}(i) \}
\]
\[
f := i \ast f ;
\]
\[
\{ f = \text{fact}(i) \}
\]
\[
\{ f = \text{fact}(n) \}
\]

\text{assert}(f = \text{fact}(n))
Example: Iterative factorial (cont.)

$$\text{ifact} \ (n : \text{Nat}) =$$

\[\text{assume}(\text{true});\]

\[f := 1;\]

\[i := 0;\]

\{f = \text{fact}(i)\}

\text{while } i \neq n \text{ do}

\{f = \text{fact}(i) \land i \neq n\} \implies

\{(i + 1) \times f = \text{fact}(i + 1)\} \iff (i + 1) \times f = (i + 1) \times \text{fact}(i)

\[i := i + 1;\]

\{i \times f = \text{fact}(i)\}

\[f := i \times f;\]

\{f = \text{fact}(i)\}

\{f = \text{fact}(n)\}

\text{assert}(f = \text{fact}(n))
Example: Iterative factorial (cont.)

\[ \text{ifact} (n : \text{Nat}) = \]
\begin{verbatim}
assume(true);
\end{verbatim}

\[ f := 1 ; \]
\begin{verbatim}
\{ f = \text{fact}(0) \} \iff \{ f = 1 \}
\end{verbatim}

\[ i := 0 ; \]
\begin{verbatim}
\{ f = \text{fact}(i) \}
\end{verbatim}

\begin{verbatim}
while \( i \neq n \) do
\{ f = \text{fact}(i) \land i \neq n \} \implies
\{ (i + 1) \ast f = \text{fact}(i + 1) \} \iff (i + 1) \ast f = (i + 1) \ast \text{fact}(i)
\end{verbatim}

\[ i := i + 1 ; \]
\begin{verbatim}
\{ i \ast f = \text{fact}(i) \}
\end{verbatim}

\[ f := i \ast f ; \]
\begin{verbatim}
\{ f = \text{fact}(i) \}
\end{verbatim}

[\{ f = \text{fact}(n) \}]
\begin{verbatim}
assert(f = \text{fact}(n))
\end{verbatim}
Example: Iterative factorial (cont.)

\[
\text{ifact}(n : \text{Nat}) = \\
\text{assume}(\text{true}); \\
\{1 = 1\} \iff \{\text{true}\} \\
f := 1; \\
\{f = \text{fact}(0)\} \iff \{f = 1\} \\
i := 0; \\
\{f = \text{fact}(i)\} \\
\text{while } i \neq n \text{ do} \\
\{f = \text{fact}(i) \land i \neq n\} \implies \\
\{(i + 1) \ast f = \text{fact}(i + 1)\} \iff (i + 1) \ast f = (i + 1) \ast \text{fact}(i) \\
i := i + 1; \\
\{i \ast f = \text{fact}(i)\} \\
f := i \ast f; \\
\{f = \text{fact}(i)\} \\
\{f = \text{fact}(n)\} \\
\text{assert}(f = \text{fact}(n))
\]
Partial correctness of the Iterative Reverse

left as an exercise ...
Partial correctness of the Iterative Sum

\[
\begin{align*}
\text{assume(true);}\\
\ell' : \text{List}[\text{Nat}] ; \\
\text{r := 0 ;} \\
\ell' := \ell ; \\
\text{while } \ell' \neq [] \text{ do} \\
\quad \text{invariant(?) ;} \\
\quad \text{r := } \text{r + head(\ell')} ; \\
\quad \ell' := \text{tail(\ell')} ; \\
\text{assert(r = } \Sigma(\ell)) \\
\end{align*}
\]
Partial correctness of the Iterative Sum

\[ r : \text{Nat} ; \]

\[ \text{isum} (\ell : \text{List}[\text{Nat}]) = \]

\[ \text{assume(true)} ; \]

\[ \ell' : \text{List}[\text{Nat}] ; \]

\[ r := 0 ; \]

\[ \ell' := \ell ; \]

\[ \text{while} \: \ell' \neq [] \: \text{do} \]

\[ \text{invariant}(r + \Sigma(\ell') = \Sigma(\ell)) ; \]

\[ r := r + \text{head}(\ell') ; \]

\[ \ell' := \text{tail}(\ell') ; \]

\[ \text{assert}(r = \Sigma(\ell)) \]
Use of ghost (auxilliary) variables

\[ r : \text{Nat} ; \]
\[ isum(\ell : \text{List}[\text{Nat}]) = \]
\[ \text{assume}(\text{true}); \]
\[ \sigma : \text{List}[\text{Nat}] ; \]
\[ \ell' : \text{List}[\text{Nat}] ; \]
\[ r := 0 ; \]
\[ \sigma := [] ; \]
\[ \ell' := \ell ; \]
\[ \text{while } \ell' \neq [] \text{ do} \]
\[ \quad \text{invariant}\left(\left(r = \Sigma(\sigma)\right) \land \left(\ell = \sigma @ \ell'\right)\right) \]
\[ \quad r := r + \text{head}(\ell') ; \]
\[ \quad \sigma := \sigma \circ \text{head}(\ell') ; \]
\[ \quad \ell' := \text{tail}(\ell') ; \]
\[ \quad \text{assert}(r = \Sigma(\ell)) \]
Proving partial correctness of isum

left as an exercise ...
Summary

- Imperative programs transform memory states. Programs can be seen as state machines.
- Assertions about states can be written in logic-based specification languages.
Summary

- Imperative programs transform memory states. Programs can be seen as state machines.
- Assertions about states can be written in logic-based specification languages.
- A program must be annotated with assertions specifying the assumptions on the initial state, the guarantees on the final state, as well as loop invariants.
- Pre-post condition reasoning allow to check that the guaranteed are indeed satisfied under the considered assumptions. This reasoning can be carried out formally in Hoare logic.
Summary

- Imperative programs transform memory states. Programs can be seen as state machines.
- Assertions about states can be written in logic-based specification languages.
- A program must be annotated with assertions specifying the assumptions on the initial state, the guarantees on the final state, as well as loop invariants.
- Pre-post condition reasoning allow to check that the guaranteed are indeed satisfied under the considered assumptions. This reasoning can be carried out formally in Hoare logic.
- Proving the validity of Hoare triples must be done in the considered theory of data.
- Such proofs can be done either manually, or semi-manually using theorem provers, or automatically in some cases using decision procedures, e.g., those implemented in SMT solvers.