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:

$$
\begin{aligned}
S ::= \;& \texttt{skip} \\
\mid \;& x := E \\
\mid \;& S \; ; \; S \\
\mid \;& \texttt{if} \;\; C \;\; \texttt{then} \;\; S \;\; \texttt{else} \;\; S \\
\mid \;& \texttt{while} \;\; C \;\; \texttt{do} \;\; S
\end{aligned}
$$

where E is a term and C is a formula over X in $FO(D, Op, Rel)$.

Example of a program

$f : Nat$;

```
ifact (n : Nat) =
    i : Nat ;
    f := 1 ;
    i := 0 ;
    while i ≠ n do
        i := i + 1 ;
        f := i * f
```

Another example of a program

```
r : Nat ;

isum (ℓ : List[Nat]) =
   ℓ′ : List[Nat] ;
   r := 0 ;
   ℓ′ := ℓ ;
   while ℓ′ ≠ [] do
           r := r + head(ℓ′) ;
           ℓ′ := tail(ℓ′)
```

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:

$$\mu : \mathsf{X} \to \mathsf{D}$$

- Transitions between states correspond to the execution of statements:

$$\mu \xrightarrow{\mathsf{S}} \mu'$$

Semantics: Transition rules

$$\frac{}{\mu \xrightarrow{\texttt{skip}} \mu} \qquad \frac{\langle \mathsf{exp} \rangle_\mu = \mathsf{d}}{\mu \xrightarrow{\mathsf{x:=exp}} \mu[\mathsf{x} \leftarrow \mathsf{d}]}$$

$$\frac{\mu \xrightarrow{\mathsf{S}_1} \nu \qquad \nu \xrightarrow{\mathsf{S}_2} \mu'}{\mu \xrightarrow{\mathsf{S}_1;\mathsf{S}_2} \mu'}$$

$$\frac{\mu \models \mathsf{C} \qquad \mu \xrightarrow{\mathsf{S}_1} \mu'}{\mu \xrightarrow{\texttt{if } \mathsf{C} \texttt{ then } \mathsf{S}_1 \texttt{ else } \mathsf{S}_2} \mu'} \qquad \frac{\mu \models \neg\mathsf{C} \qquad \mu \xrightarrow{\mathsf{S}_2} \mu'}{\mu \xrightarrow{\texttt{if } \mathsf{C} \texttt{ then } \mathsf{S}_1 \texttt{ else } \mathsf{S}_2} \mu'}$$

$$\frac{\mu \models \neg\mathsf{C}}{\mu \xrightarrow{\texttt{while } \mathsf{C} \texttt{ do } \mathsf{S}} \mu} \qquad \frac{\mu \models \mathsf{C} \qquad \mu \xrightarrow{\mathsf{S}} \nu \qquad \nu \xrightarrow{\texttt{while } \mathsf{C} \texttt{ do } \mathsf{S}} \mu'}{\mu \xrightarrow{\texttt{while } \mathsf{C} \texttt{ do } \mathsf{S}} \mu'}$$

Assertions

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

```
f : Nat ;

ifact (n : Nat) =
    assume(true);
    i : Nat ;
    f := 1 ;
    i := 0 ;
    while i ≠ n do
        i := i + 1 ;
        f := i * f ;
    assert(f = fact(n))
```

Assertions

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

r : Nat ;

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

Assertions

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

r : Nat ;

isum (ℓ : List[Nat]) $=$
    assume($\forall e \in \star.\ In(e, \ell) \Rightarrow (e = 1)$)
    ℓ' : List[Nat] ;
    x := 0 ;
    $\ell' := \ell$;
    while $\ell' \neq []$ do
        r := r + head(ℓ') ;
        ℓ' := tail(ℓ') ;
    assert($r = |\ell|$)

Assume – Assert statements: Semantics

- Let $\perp$ be a special *error* state
- Transition rules:

$$\frac{\mu \models \phi}{\mu \xrightarrow{\texttt{assume}(\phi)} \mu}$$

$$\frac{\mu \models \phi}{\mu \xrightarrow{\texttt{assert}(\phi)} \mu} \qquad \frac{\mu \models \neg\phi}{\mu \xrightarrow{\texttt{assert}(\phi)} \perp}$$

Loop Invariants

```
f : Nat ;

ifact (n : Nat) =
    assume(true);
    i : Nat ;
    f := 1 ;
    i := 0 ;
    while i ≠ n do
        invariant(?);
        i := i + 1 ;
        f := i * f ;
    assert(f = fact(n))
```

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

Loop Invariants

```
f : Nat ;

ifact (n : Nat)  =
    assume(true);
    i : Nat ;
    f := 1 ;
    i := 0 ;
    while i ≠ n  do
        invariant(?);
        i := i + 1 ;
        f := i ∗ f ;
    assert(f = 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 : Nat ;

ifact (n : Nat)  =
    assume(true);
    i : Nat ;
    f := 1 ;
    i := 0 ;
    while i ≠ n  do
          invariant(?);
          i := i + 1 ;
          f := i ∗ f ;
    assert(f = 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 : Nat ;

ifact (n : Nat) =
    assume(true);
    i : Nat ;
    f := 1 ;
    i := 0 ;
    while i ≠ n do
        invariant(f = fact(i));
        i := i + 1 ;
        f := i * f ;
    assert(f = fact(n))
```

- *A property that is true initially, and after each iteration.*
- But there are many invariants!!: true, $f \geq 1$, ...
- A "useful invariant":
  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

ρ : List[$\star$] ;

irev (ℓ : List[$\star$]) $=$
    assume(true);

    assert($\rho = $ Rev(ℓ))

Example: Reversing a list

ρ : List[$\star$] ;

irev (ℓ : List[$\star$]) $=$

assume(true);
ℓ' : List[$\star$] ;
ρ := [] ; % *ρ is the reverse of the treated prefix of ℓ*
ℓ' := ℓ ; % *ℓ' is the non-treated suffix of ℓ*
while do

assert($\rho = \text{Rev}(\ell)$)

Example: Reversing a list

ρ : List[$\star$] ;

irev (ℓ : List[$\star$]) =

assume(true);

ℓ' : List[$\star$] ;

$\rho := []$; % *ρ is the reverse of the treated prefix of ℓ*

$\ell' := \ell$; % *ℓ' is the non-treated suffix of ℓ*

while do

invariant($\ell = \text{Rev}(\rho)@\ell'$)

assert($\rho = \text{Rev}(\ell)$)

Example: Reversing a list

```
ρ : List[⋆] ;

irev (ℓ : List[⋆]) =
    assume(true);
    ℓ′ : List[⋆] ;
    ρ := [] ;       % ρ is the reverse of the treated prefix of ℓ
    ℓ′ := ℓ ;       % ℓ′ is the non-treated suffix of ℓ
    while ℓ′ ≠ []   do
        invariant(ℓ = Rev(ρ)@ℓ′)
        ρ := head(ℓ′) · ρ ;
        ℓ′ := tail(ℓ′) ;
    assert(ρ = Rev(ℓ))
```

Pre-post condition reasoning

- Consider formulas of the form:

$$\{\phi\}\ \mathsf{S}\ \{\psi\}$$

  where S is a statement, and ϕ and ψ are assertions.

- ϕ is the pre-condition, and ψ is the post-condition.

Pre-post condition reasoning

- Consider formulas of the form:

$$\{\phi\}\ \mathsf{S}\ \{\psi\}$$

  where S is a statement, and ϕ and ψ are assertions.

- ϕ is the pre-condition, and ψ is the post-condition.

- Formal Semantics:

$$\{\phi\}\ \mathsf{S}\ \{\psi\} \text{ iff } \forall \mu, \mu'. (\mu \models \phi \wedge \mu \xrightarrow{\mathsf{S}} \mu') \Rightarrow \mu' \models \psi$$

- Intuitive meaning:

  Starting from a state satisfying ϕ, if the execution of S *terminates, then the reached state must satisfy ψ.*

Pre-post condition reasoning

- Consider formulas of the form:

$$\{\phi\} \; \mathsf{S} \; \{\psi\}$$

where S is a statement, and ϕ and ψ are assertions.

- ϕ is the pre-condition, and ψ is the post-condition.
- Formal Semantics:

$$\{\phi\} \; \mathsf{S} \; \{\psi\} \;\text{ iff }\; \forall \mu, \mu'. \, (\mu \models \phi \wedge \mu \xrightarrow{\mathsf{S}} \mu') \Rightarrow \mu' \models \psi$$

- Intuitive meaning:

Starting from a state satisfying ϕ, if the execution of S *terminates, then the reached state must satisfy ψ.*

- Problem: How to prove the validity of such formulas ?

A Formal System: Hoare Logic

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

$$\frac{}{\text{Axiom}} \qquad\qquad \frac{\text{Premise}_1 \quad \cdots \quad \text{Premise}_N}{\text{Conclusion}}$$

A Formal System: Hoare Logic

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

$$\frac{}{\text{Axiom}} \qquad\qquad \frac{\text{Premise}_1 \quad \cdots \quad \text{Premise}_\text{N}}{\text{Conclusion}}$$

- Compositional reasoning using the structure of the programs:

$$\frac{\{\phi_1\}\,\mathsf{S}_1\,\{\psi_1\} \quad \cdots \quad \{\phi_\mathsf{N}\}\,\mathsf{S}_\mathsf{N}\,\{\psi_\mathsf{N}\}}{\{\phi\}\,\mathsf{Comp}(\mathsf{S}_1,\ldots,\mathsf{S}_\mathsf{N})\,\{\psi\}}$$

Hoare Logic: Axioms for Basic Statements

$$\overline{\{\phi\}\ \texttt{skip}\ \{\phi\}}$$

Hoare Logic: Axioms for Basic Statements

$$\frac{}{\{\phi\}\ \texttt{skip}\ \{\phi\}}$$

$$\frac{}{\{\phi[\mathsf{exp}/\mathsf{x}]\}\ \mathsf{x} := \mathsf{exp}\ \{\phi\}}$$

Hoare Logic: Axioms for Basic Statements

$$\overline{\{\phi\}\ \texttt{skip}\ \{\phi\}}$$

$$\overline{\{\phi[\mathsf{exp}/\mathsf{x}]\}\ \mathsf{x} := \mathsf{exp}\ \{\phi\}}$$

$$??\quad \mathsf{x} := \mathsf{x} + 2\quad \{\mathsf{x} \geq 5 \land \mathsf{x} \leq \mathsf{y} + 1\}$$

Hoare Logic: Axioms for Basic Statements

$$\frac{}{\{\phi\}\ \texttt{skip}\ \{\phi\}}$$

$$\frac{}{\{\phi[\mathsf{exp}/\mathsf{x}]\}\ \mathsf{x} := \mathsf{exp}\ \{\phi\}}$$

$$\begin{array}{rcl} ?? & \mathsf{x} := \mathsf{x}+2 & \{\mathsf{x} \geq 5 \wedge \mathsf{x} \leq \mathsf{y}+1\} \\ \{\mathsf{x}+2 \geq 5 \wedge \mathsf{x}+2 \leq \mathsf{y}+1\} & \mathsf{x} := \mathsf{x}+2 & \{\mathsf{x} \geq 5 \wedge \mathsf{x} \leq \mathsf{y}+1\} \\ \{\mathsf{x} \geq 3 \wedge \mathsf{x}+1 \leq \mathsf{y}\} & \mathsf{x} := \mathsf{x}+2 & \{\mathsf{x} \geq 5 \wedge \mathsf{x} \leq \mathsf{y}+1\} \end{array}$$

Forward version of the assignment axiom?

- Let M be a set of program states ($M \subseteq [X \to D]$), and let S be a program statement.
- Sets of immediate successors and predecessors:

$$\begin{aligned} \mathsf{post}(M, S) &= \{\mu' \,:\, \exists \mu \in M.\ \mu \xrightarrow{S} \mu'\} \\ \mathsf{pre}(M, S) &= \{\mu \,:\, \exists \mu' \in M.\ \mu \xrightarrow{S} \mu'\} \end{aligned}$$

- Let $\phi(X)$ be an assertion over X such that $[\![\phi]\!] = M$. Assertions for $\mathsf{post}(M, x := \mathsf{exp}(X))$ and $\mathsf{pre}(M, x := \mathsf{exp}(X))$?

Forward version of the assignment axiom? (cont.)

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

$$\begin{aligned}\mathrm{pre}(\phi, x := \exp)(X) &= \exists X'.\ (\phi(X') \wedge X' = \exp(X)) \\ \mathrm{post}(\phi, x := \exp)(X) &= \exists X'.\ (\phi(X') \wedge X = \exp(X'))\end{aligned}$$

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

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

- Can we do the same for the post formula?

$$\begin{aligned}\mathrm{post}(2 \leq x \wedge x \leq y, x := y) &= \exists x'.\ (2 \leq x' \wedge x' \leq y \wedge x = y) \\ &= 2 \leq y \wedge x = y\end{aligned}$$

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

Hoare Logic: Sequential composition

$$\frac{\{\phi_1\}\ \mathsf{S}_1\ \{\phi_2\} \qquad \{\phi_2\}\ \mathsf{S}_2\ \{\phi_3\}}{\{\phi_1\}\ \mathsf{S}_1;\mathsf{S}_2\ \{\phi_3\}}$$

Example: Swap

$$t := x\ ;$$

$$x := y\ ;$$

$$y := t$$

Example: Swap

$$t := x \; ;$$

$$x := y \; ;$$

$$y := t$$
$$\{x = a \wedge y = b\}$$

Example: Swap

$$t := x\,;$$

$$x := y\,;$$
$$\{x = a \wedge b = t\}$$
$$y := t$$
$$\{x = a \wedge y = b\}$$

Example: Swap

$$t := x\ ;$$
$$\{y = a \wedge b = t\}$$
$$x := y\ ;$$
$$\{x = a \wedge b = t\}$$
$$y := t$$
$$\{x = a \wedge y = b\}$$

Example: Swap

$$\{y = a \wedge b = x\}$$
$$t := x\ ;$$
$$\{y = a \wedge b = t\}$$
$$x := y\ ;$$
$$\{x = a \wedge b = t\}$$
$$y := t$$
$$\{x = a \wedge y = b\}$$

Hoare Logic: Implication rule

$$\frac{\phi_1 \Rightarrow \phi_1' \qquad \{\phi_1'\}\ \mathsf{S}\ \{\phi_2'\} \qquad \phi_2' \Rightarrow \phi_2}{\{\phi_1\}\ \mathsf{S}\ \{\phi_2\}}$$

Hoare Logic: Conditional rule

$$\frac{\{\phi \wedge \mathsf{C}\}\ \mathsf{S}_1\ \{\phi'\} \qquad \{\phi \wedge \neg \mathsf{C}\}\ \mathsf{S}_2\ \{\phi'\}}{\{\phi\}\ \texttt{if}\ \mathsf{C}\ \texttt{then}\ \mathsf{S}_1\ \texttt{else}\ \mathsf{S}_2\ \{\phi'\}}$$

Example: Minimum of 2 different values

- We want to establish:

$$\{\text{true}\}$$
$$\texttt{if}\ \ x < y\ \texttt{then}\ m := x\ \texttt{else}\ m := y$$
$$\{m \leq x \wedge m \leq y\}$$

Example: Minimum of 2 different values

- We want to establish:

$$\{true\}$$
$$\texttt{if}\ \ x < y\ \texttt{then}\ m := x\ \texttt{else}\ m := y$$
$$\{m \leq x \wedge m \leq y\}$$

- Premises that must be proved:
  1. $\{x < y\}\ m := x\ \{m \leq x \wedge m \leq y\}$
  2. $\{y < x\}\ m := y\ \{m \leq x \wedge m \leq y\}$

Example: Minimum of 2 different values

- We want to establish:

$$\{true\}$$
$$\texttt{if}\ \ x < y\ \texttt{then}\ m := x\ \texttt{else}\ m := y$$
$$\{m \leq x \wedge m \leq y\}$$

- Premises that must be proved:
  1. $\{x < y\}\ m := x\ \{m \leq x \wedge m \leq y\}$
  2. $\{y < x\}\ m := y\ \{m \leq x \wedge m \leq y\}$

- Proof of Premise 1: Assignment axiom + implication rule
  - $\{x \leq x \wedge x \leq y\}\ m := x\ \{m \leq x \wedge m \leq y\}$
  - $x < y \Rightarrow x \leq y$

Example: Minimum of 2 different values

- We want to establish:

$$\{true\}$$
$$\texttt{if } x < y \texttt{ then } m := x \texttt{ else } m := y$$
$$\{m \leq x \wedge m \leq y\}$$

- Premises that must be proved:
  1. $\{x < y\}\ m := x\ \{m \leq x \wedge m \leq y\}$
  2. $\{y < x\}\ m := y\ \{m \leq x \wedge m \leq y\}$

- Proof of Premise 1: Assignment axiom + implication rule
  - $\{x \leq x \wedge x \leq y\}\ m := x\ \{m \leq x \wedge m \leq y\}$
  - $x < y \Rightarrow x \leq y$

- Proof of Premise 2 is identical.

Hoare Logic: Iteration rule

$$\frac{\{\phi \wedge C\}\ S\ \{\phi\}}{\{\phi\}\ \texttt{while}\ C\ \texttt{do}\ S\ \{\phi \wedge \neg C\}}$$

Example: Iterative factorial

- Assignment + Sequential composition rules:

$$\{(i+1) * f = fact(i+1)\}$$
$$i := i+1\ ;$$
$$\{i * f = fact(i)\}$$
$$f := i * f\ ;$$
$$\{f = fact(i)\}$$

Example: Iterative factorial

- Assignment + Sequential composition rules:

$$\{(i+1)*f = fact(i+1)\}$$
$$i := i+1\ ;$$
$$\{i*f = fact(i)\}$$
$$f := i*f\ ;$$
$$\{f = fact(i)\}$$

- Definition of *fact*: $fact(i+1) = (i+1)*fact(i)$

Example: Iterative factorial

- Assignment + Sequential composition rules:

$$\{(i+1)*f = fact(i+1)\}$$
$$i := i+1\ ;$$
$$\{i*f = fact(i)\}$$
$$f := i*f\ ;$$
$$\{f = fact(i)\}$$

- Definition of *fact*: $fact(i+1) = (i+1)*fact(i)$
- Theory of integers: $f = fact(i)) \implies (i+1)*f = (i+1)*fact(i)$

Example: Iterative factorial

- Assignment + Sequential composition rules:

$$\{(i+1)*f = fact(i+1)\}$$
$$i := i+1\ ;$$
$$\{i*f = fact(i)\}$$
$$f := i*f\ ;$$
$$\{f = fact(i)\}$$

- Definition of *fact*: $fact(i+1) = (i+1)*fact(i)$

- Theory of integers: $f = fact(i)) \implies (i+1)*f = (i+1)*fact(i)$

- Implication rule:

$$\{(f = fact(i))\}$$
$$i := i+1\ ;\ f := i*f$$
$$\{(f = fact(i))\}$$

Example: Iterative factorial (cont.)

- So far:

$$\{f = fact(i) \qquad\qquad \}$$
$$i := i + 1 \; ; \; f := i * f$$
$$\{f = fact(i)\}$$

Example: Iterative factorial (cont.)

- So far: + Implication rule

$$\{f = fact(i) \wedge i \neq n\}$$
$$i := i + 1 \; ; \; f := i * f$$
$$\{f = fact(i)\}$$

Example: Iterative factorial (cont.)

- So far: + Implication rule

$$\{f = fact(i) \wedge i \neq n\}$$
$$i := i + 1 \; ; \; f := i * f$$
$$\{f = fact(i)\}$$

- Iteration rule:

$$\{f = fact(i)\}$$
$$\texttt{while } (i \neq n) \texttt{ do } \{i := i + 1 \; ; \; f := i * f\}$$
$$\{f = fact(i) \wedge i = n\}$$

Example: Iterative factorial (cont.)

- So far: + Implication rule

$$\{f = fact(i) \wedge i \neq n\}$$
$$i := i + 1\ ;\ f := i * f$$
$$\{f = fact(i)\}$$

- Iteration rule: + Implication rule

$$\{f = fact(i)\}$$
$$\texttt{while}\ (i \neq n)\ \texttt{do}\ \{i := i + 1\ ;\ f := i * f\}$$
$$\{f = fact(i) \wedge i = n\}$$
$$\Longrightarrow$$
$$\{f = fact(n)\}$$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat) \;=$

$assume(true);$

$f := 1\;;$

$i := 0\;;$

`while` $i \neq n$ `do`

$i := i + 1\;;$

$f := i * f\;;$

$assert(f = fact(n))$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat)\ =$

$assume(true);$

$f := 1\ ;$

$i := 0\ ;$

`while` $i \neq n$ `do`

$i := i + 1\ ;$

$f := i * f\ ;$

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

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

$assert(f = fact(n))$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat) \;=$

$\quad assume(true);$

$\quad f := 1\;;$

$\quad i := 0\;;$

$\quad$ `while` $i \neq n$ `do`

$$\{(i+1) * f \;=\; fact(i+1)\} \iff (i+1) * f \;=\; (i+1) * fact(i)$$

$$i := i + 1\;;$$

$$\{i * f \;=\; fact(i)\}$$

$$f := i * f\;;$$

$$\{f \;=\; fact(i)\}$$

$\quad \{f \;=\; fact(n)\}$

$\quad assert(f \;=\; fact(n))$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat) \; =$

$\quad assume(true);$

$\quad f := 1\;;$

$\quad i := 0\;;$

$\quad$ `while` $i \neq n$ `do`

$$\{f = fact(i) \wedge i \neq n\} \implies$$

$$\{(i+1)*f = fact(i+1)\} \iff (i+1)*f = (i+1)*fact(i)$$

$$i := i+1\;;$$

$$\{i*f = fact(i)\}$$

$$f := i*f\;;$$

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

$\quad \{f = fact(n)\}$

$\quad assert(f = fact(n))$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat) \ =$

$assume(true);$

$f := 1\ ;$

$i := 0\ ;$

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

$\texttt{while}\ i \neq n\ \texttt{do}$

$\{f = fact(i) \wedge i \neq n\} \implies$

$\{(i+1) * f = fact(i+1)\} \iff (i+1) * f = (i+1) * fact(i)$

$i := i + 1\ ;$

$\{i * f = fact(i)\}$

$f := i * f\ ;$

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

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

$assert(f = fact(n))$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat) \; =$

$\quad assume(true);$

$\quad f := 1\;;$

$\quad \{f = fact(0)\} \iff \{f = 1\}$

$\quad i := 0\;;$

$\quad \{f = fact(i)\}$

$\quad \texttt{while}\ i \neq n\ \texttt{do}$

$\qquad \{f = fact(i) \wedge i \neq n\} \implies$

$\qquad \{(i+1) * f = fact(i+1)\} \iff (i+1) * f = (i+1) * fact(i)$

$\qquad i := i + 1\;;$

$\qquad \{i * f = fact(i)\}$

$\qquad f := i * f\;;$

$\qquad \{f = fact(i)\}$

$\quad \{f = fact(n)\}$

$\quad assert(f = fact(n))$

Example: Iterative factorial (cont.)

$ifact\,(n : Nat) \; =$

$\quad assume(true);$

$\quad \{1 = 1\} \iff \{true\}$

$\quad f := 1 \, ;$

$\quad \{f = fact(0)\} \iff \{f = 1\}$

$\quad i := 0 \, ;$

$\quad \{f = fact(i)\}$

$\quad \texttt{while } i \neq n \texttt{ do}$

$\qquad \{f = fact(i) \wedge i \neq n\} \implies$

$\qquad \{(i+1) * f = fact(i+1)\} \iff (i+1) * f = (i+1) * fact(i)$

$\qquad i := i + 1 \, ;$

$\qquad \{i * f = fact(i)\}$

$\qquad f := i * f \, ;$

$\qquad \{f = fact(i)\}$

$\quad \{f = fact(n)\}$

$\quad assert(f = fact(n))$

Partial correctness of the Iterative Reverse

left as an exercise ...

Partial correctness of the Iterative Sum

```
r : Nat ;
isum(ℓ : List[Nat]) =
    assume(true);
    ℓ′ : List[Nat] ;
    r := 0 ;
    ℓ′ := ℓ ;
    while ℓ′ ≠ [] do
          invariant(?);
          r := r + head(ℓ′) ;
          ℓ′ := tail(ℓ′) ;
    assert(r = Σ(ℓ))
```

Partial correctness of the Iterative Sum

$r : Nat$;

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

Use of ghost (auxilliary) variables

$r : Nat$;

$isum(\ell : List[Nat]) =$
$\quad assume(true);$
$\quad \sigma : List[Nat]$;
$\quad \ell' : List[Nat]$;
$\quad r := 0$;
$\quad \sigma := []$;
$\quad \ell' := \ell$;
$\quad while\ \ell' \neq []\ \ do$
$\qquad invariant((r = \Sigma(\sigma)) \wedge (\ell = \sigma @ \ell'))$
$\qquad r := r + head(\ell')$;
$\qquad \sigma := \sigma \circ head(\ell')$;
$\qquad \ell' := tail(\ell')$;
$\quad assert(r = \Sigma(\ell))$

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.