Theme 2: Verifying Imperative Sequential Programs

Lecture 4:
Partial Correctness of Imperative Programs – Hoare Logic

Matthieu Sozeau
Inria & 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 quantifier-free formula over $X$ in $FO(D, Op, Rel)$.

Example of a program

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

Another example of a program

```plaintext
r : Nat;
isum (ℓ : List[Nat]) =
  ℓ : List[Nat];
r := 0;
ℓ' := ℓ;
while ℓ' \neq [] 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:
  \[ \mu : X \to D \]

Transitions between states correspond to the execution of statements:
\[ \mu \xrightarrow{S} \mu' \]

Semantics: Transition rules

\[
\begin{align*}
\mu \xrightarrow{\text{skip}} \mu \\
\mu \xrightarrow{x := \text{exp}} \mu[x := d] \\
\mu \xrightarrow{S_1 ; S_2} \mu' \\
\mu \xrightarrow{\text{if } C \text{ then } S_1 \text{ else } S_2} \mu' \\
\mu \xrightarrow{\text{while } C \text{ do } S} \mu' \\
\mu \models C \\
\mu \models \neg C
\end{align*}
\]

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.

f : Nat ;
ifact (n : Nat) =
  assume(true);
  i := 0 ;
  while i \neq n do
    i := i + 1 ;
    f := i * f ;
  assert(f = \text{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.

r : Nat ;
isum (\ell : \text{List[Nat]}) =
  assume(true);
  r := 0 ;
  \ell' := \ell ;
  while \ell' \neq [] do
    r := r + \text{head}(\ell') ;
    \ell' := \text{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(φ)` and `assert(φ)` where φ is a FO formula over X.

```plaintext
r : Nat ;

isum (ℓ : List[Nat]) =
  assume(∀ e ∈ ⋆. ln(e, ℓ) ⇒ (e = 1))

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

**Assume – Assert statements: Semantics**

- Let ⊥ be a special error state.
- Transition rules:

```plaintext
\[
\frac{\mu \models φ}{\mu \overset{assume(φ)}{\longrightarrow} \mu}
\]
\[
\frac{\mu \models φ}{\mu \overset{assert(φ)}{\longrightarrow} \mu}
\]
\[
\frac{\mu \models \neg φ}{\mu \overset{assert(φ)}{\longrightarrow} \perp}
\]
```

**Loop Invariants**

```plaintext
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 ≥ 0, f ≥ 1, ...
- A “useful invariant”:
  After the last iteration, it implies the desired post-condition.

```plaintext
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 ≥ 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

\[\rho : \text{List}\,\{\,\cdot\,\}\,;\]
\[\text{irev}\,(\ell : \text{List}\,\{\,\cdot\,\}) =\]
\[\begin{align*}
&\text{assume(true);} \\
&\ell' : \text{List}\,\{\,\cdot\,\}; \\
&\rho := []; \quad \% \rho \text{ is the reverse of the treated prefix of } \ell \\
&\ell' := \ell; \quad \% \ell' \text{ is the non-treated suffix of } \ell \\
&\text{while } \ell' \neq [] \text{ do} \\
&\quad \text{invariant}(\ell = \text{Rev}(\rho) \square \ell') \\
&\quad \rho := \text{head}(\ell') \cdot \rho; \\
&\quad \ell' := \text{tail}(\ell'); \\
&\quad \text{assert(}\rho = \text{Rev}(\ell))
\end{align*}\]

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 \(\psi\).
- Problem: How to prove the validity of such formulas?

A Formal System: Hoare Logic

- A set of axioms and inference rules of the form:
  \[
  \begin{array}{c}
  \text{Axiom} \\
  \text{Premise}_1 \quad \cdots \quad \text{Premise}_N \\
  \downarrow \quad \Downarrow \\
  \text{Conclusion}
  \end{array}
  \]
- Compositional reasoning using the structure of the programs:
  \[
  \begin{align*}
  \{\phi_1\} \,S_1\, \{\psi_1\} & \quad \cdots \quad \{\phi_N\} \,S_N\, \{\psi_N\} \\
  \{\phi\} \text{Comp} (S_1, \ldots, S_N) \,\{\psi\}
  \end{align*}
  \]
Hoare Logic: Axioms for Basic Statements

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

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

Forward version of the assignment axiom?

The assignment rule goes backwards!

What’s wrong with this?

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

We could deduce:

\[ \{ x = 5 \} x := 3 \{ 3 = 5 \} \]

Wrong! How about this?

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

We could deduce:

\[ \{ x = 5 \} x := x + 1 \{ x = 5 \land x = x + 1 \} \]

Wrong! What is true before an assignment is not necessarily true after the assignment: assignment is a destructive operation.

Forward version of the assignment axiom? (cont.)

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

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

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

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

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

- 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., \( FO(\mathbb{N}, \{0, 1, +\}, \{\leq\}) \). Not always possible / expensive.
Hoare Logic: Sequential composition

\[
\begin{array}{c}
\{ \phi_1 \} S_1 \{ \phi_2 \} \\
\{ \phi_2 \} S_2 \{ \phi_3 \}
\end{array}
\Rightarrow

\{ \phi_1 \} S_1 ; S_2 \{ \phi_3 \}
\]

Example: Swap

\[
\begin{align*}
\{ y = a \land x = b \} \\
t := x ; \\
\{ y = a \land t = b \} \\
x := y ; \\
\{ x = a \land t = b \} \\
y := t \\
\{ x = a \land y = b \}
\end{align*}
\]

Hoare Logic: Implication rule

\[
\begin{array}{c}
\phi_1 \Rightarrow \phi'_1 \\
\{ \phi'_1 \} S \{ \phi'_2 \} \\
\phi'_2 \Rightarrow \phi_2
\end{array}
\Rightarrow

\{ \phi_1 \} S \{ \phi_2 \}
\]

Hoare Logic: Conditional rule

\[
\begin{array}{c}
\{ \phi \land C \} S_1 \{ \phi' \} \\
\{ \phi \land \neg C \} S_2 \{ \phi' \}
\end{array}
\Rightarrow

\{ \phi \text{ if } C \text{ then } S_1 \text{ else } S_2 \{ \phi' \} \}
Example: Minimum of 2 different values

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

- Premises that must be proved:
  \[
  \begin{align*}
    \{x < y\} & m := x \{m \leq x \land m \leq y\} \\
    \{y \leq x\} & m := y \{m \leq x \land m \leq y\}
  \end{align*}
  \]

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

- Proof of Premise 2:
  - Assignment: \(\{y \leq x\} \quad m := y \quad \{m \leq x \land m \leq y\}\)
  - Implication \(y \leq x \Rightarrow y \leq x \land y \leq y\)

Example: Iterative factorial

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

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

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

- Implication rule:
  \[
  \begin{align*}
    \{(f = \text{fact}(i))\} & \\
    i := i + 1; & \\
    f := i \ast f & \\
    \{(f = \text{fact}(i))\}
  \end{align*}
  \]

Example: Iterative factorial (cont.)

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

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

```plaintext
ifact(n : Nat) =
  assume(true);
  {1 = 1} \iff\ {true}
f := 1;
{f = fact(0)} \iff\ {f = 1}
i := 0;
{f = fact(i)}
while i \neq n do
  {f = fact(i) \land i \neq n} \implies\n  {(i + 1) \ast f = fact(i + 1)} \iff\n  (i + 1) \ast f = (i + 1) \ast fact(i)
i := i + 1;
{f = fact(i)}
f := i \ast f;
{f = fact(i)}
{f = fact(n)}
assert(f = fact(n))
```

Partial correctness of the Iterative Sum

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

Left as an exercise ...

Partial correctness of the Iterative Reverse

```plaintext
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

\[
\begin{align*}
\text{r} &: \text{Nat} ; \\
\text{isum}(\ell : \text{List}[\text{Nat}]) &= \\
&\text{assume(true);} \\
&\sigma : \text{List}[\text{Nat}] ; \\
&\ell' : \text{List}[\text{Nat}] ; \\
&\text{r} := 0 ; \\
&\sigma := [] ; \\
&\ell' := \ell ; \\
&\text{while} \ell' \neq [] \text{ do} \\
&\quad \text{invariant}((\text{r} = \Sigma(\sigma)) \land (\ell = \sigma@\ell')) \\
&\quad \text{r} := \text{r} + \text{head}(\ell') ; \\
&\quad \sigma := \sigma \circ \text{head}(\ell') ; \\
&\quad \ell' := \text{tail}(\ell') ; \\
&\quad \text{assert}(\text{r} = \Sigma(\ell))
\end{align*}
\]

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.
- 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 guarantees 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.