COMP 507: Computer-Aided Program Design

Fall 2014

Swarat Chaudhuri

April 7, 2015

Goal: Correctness proofs

Prove that an algorithm written in an imperative language is correct

Induction for algorithmic correctness

Induction for **functional programs**:

- The program computes the right outputs on the empty list
- Assuming the program computes the right outputs on a list L, it computes the right outputs on $m :: L$

Induction for algorithmic correctness

Induction for **functional programs**:

- The program computes the right outputs on the empty list
- Assuming the program computes the right outputs on a list L, it computes the right outputs on $m :: L$

Induction for **imperative programs**:

- All program executions of length 1 are correct
- If all executions of length k lead to correct outputs, then so do all executions of length $(k + 1)$

How do we know this program is correct?

```
method Find(a: array<int>, x: int) returns (j : int)
requires a != null;
{
   var m := 0;
   var n := a.Length;
   while (m < n)
   {
       j :=  (m + n) / 2;
       if (a[j] < x) {
           m :=  j + 1;
        } else if (x < a[j]) {
           n :=  j;
        } else {
            return;
        }
    }
   j := -1;
}
```

Correctness of programs

- **Operational semantics** defines how a program *actually* behaves
- **Specifications** state how a program *should* behave
- **Goal:** Guarantee that the program follows the specification

Language syntax (can be augmented with other constructs):
Assume a set of *variables Var*, a set of constants *Const*, a set of *arithmetic operators Func*, a set of *relational operators Pred*

$$
\begin{aligned}
e &::= \mathtt{f}(e_1, e_2) \mid x \mid \mathtt{n} \qquad \text{where } x \in \mathit{Var}, \mathtt{n} \in \mathit{Const}, \mathtt{f} \in \mathit{Func} \\
b &::= \mathtt{R}(e_1, e_2) \qquad \text{where } \mathtt{R} \in \mathit{Pred} \\
S &::= x := e \mid S_1; S_2 \mid \textbf{skip} \mid \\
&\quad\ \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } b \textbf{ do } S_1
\end{aligned}
$$

Example:

```
i := 0;
sum := 0;
while (i > n) {
   i := i + 1;
   sum := sum + i;
}
```

How are expressions/statements evaluated?

Operational semantics

How are expressions/statements evaluated?

Interpretation:

- Domain of *values* over which variables range (specifically, integers)
- Each constant `a` mapped to a value a
- Each arithmetic operator `f` mapped to a mathematical function that returns a value
  - $+$ mapped to the addition operation over integers
- Each relational operator `R` mapped to a mathematical predicate
  - $>=$ mapped to operator $\geq$ over integers

Operational semantics

How are expressions/statements evaluated?

Interpretation:

- Domain of *values* over which variables range (specifically, integers)
- Each constant `a` mapped to a value *a*
- Each arithmetic operator `f` mapped to a mathematical function that returns a value
  - $+$ mapped to the addition operation over integers
- Each relational operator `R` mapped to a mathematical predicate
  - $>=$ mapped to operator $\geq$ over integers

State σ: function mapping variables to values

Semantic judgments

What does an expression or statement do?

- $\langle e, \sigma \rangle \rightsquigarrow n$: Arithmetic expression e evaluates to value n at state σ
- $\langle b, \sigma \rangle \rightsquigarrow bv$: Boolean expression b evaluates to value bv at state σ
- $\langle S, \sigma \rangle \rightsquigarrow \sigma'$: If you start executing S at state σ, then if the program terminates, then you end up at state σ'

Defining semantics

Semantics usually presented using *inference rules*:

$$\frac{J_1 \qquad \ldots \qquad J_k}{J} \qquad (J \text{ holds assuming } J_1 \text{ through } J_k \text{ hold})$$

Defining semantics (1)

- $\langle \mathtt{a}, \sigma \rangle \rightsquigarrow a$
- $\langle x, \sigma \rangle \rightsquigarrow \sigma(x)$
- $\dfrac{\langle e_1, \sigma \rangle \rightsquigarrow a_1 \qquad \langle e_2, \sigma \rangle \rightsquigarrow a_2}{\langle \mathtt{f}(e_1, e_2), \sigma \rangle \rightsquigarrow f(a_1, a_2)}$
- $\dfrac{\langle e_1, \sigma \rangle \rightsquigarrow a_1 \qquad \langle e_2, \sigma \rangle \rightsquigarrow a_2}{\langle \mathtt{R}(e_1, e_2), \sigma \rangle \rightsquigarrow R(a_1, a_2)}$

Defining semantics (2)

- $\langle \textbf{skip}, \sigma \rangle \leadsto \sigma$
- $$\frac{\langle e, \sigma \rangle \leadsto a}{\langle x := e, \sigma \rangle \leadsto \sigma[x \mapsto a]}$$
- $$\frac{\langle S_1, \sigma \rangle \leadsto \sigma' \qquad \langle S_2, \sigma' \rangle \leadsto \sigma''}{\langle S_1; S_2, \sigma \rangle \leadsto \sigma''}$$
- $$\frac{\langle b, \sigma \rangle \leadsto \mathit{true} \qquad \langle S_1, \sigma \rangle \leadsto \sigma'}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, \sigma \rangle \leadsto \sigma'}$$
- $$\frac{\langle b, \sigma \rangle \leadsto \mathit{false} \qquad \langle S_2, \sigma \rangle \leadsto \sigma'}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2, \sigma \rangle \leadsto \sigma'}$$
- $$\frac{\langle b, \sigma \rangle \leadsto \mathit{false}}{\langle \textbf{while } b \textbf{ do } S_1, \sigma \rangle \leadsto \sigma}$$
- $$\frac{\langle b, \sigma \rangle \leadsto \mathit{true} \qquad \langle S_1; \textbf{while } b \textbf{ do } S_1, \sigma \rangle \leadsto \sigma'}{\langle \textbf{while } b \textbf{ do } S_1, \sigma \rangle \leadsto \sigma'}$$

Operational semantics: Notes

Executions: Derivation trees for judgments

Evaluation order: The rules can be applied any any order

- In $(e_1 + e_2)$, we could evaluate e_1 first or e_2 first
- The final result is the same

Operational semantics: Notes

Executions: Derivation trees for judgments

Evaluation order: The rules can be applied any any order

- In $(e_1 + e_2)$, we could evaluate e_1 first or e_2 first
- The final result is the same

Termination: Is it possible that there's no σ' such that $\langle P, \sigma \rangle \rightsquigarrow \sigma'$?

Executions: Derivation trees for judgments

Evaluation order: The rules can be applied any any order

- In $(e_1 + e_2)$, we could evaluate e_1 first or e_2 first
- The final result is the same

Termination: Is it possible that there's no σ' such that $\langle P, \sigma \rangle \rightsquigarrow \sigma'$?

Correctness: Program guarantees a property *Post* under the assumption *Pre* if

> *For all σ, σ' such that σ satisfies Pre and $\langle P, \sigma \rangle \rightsquigarrow \sigma'$, it is guaranteed that σ' satisfies Post*

Operational semantics: Notes

Executions: Derivation trees for judgments

Evaluation order: The rules can be applied any any order

- In $(e_1 + e_2)$, we could evaluate e_1 first or e_2 first
- The final result is the same

Termination: Is it possible that there's no σ' such that $\langle P, \sigma \rangle \rightsquigarrow \sigma'$?

Correctness: Program guarantees a property *Post* under the assumption *Pre* if

> *For all σ, σ' such that σ satisfies Pre and $\langle P, \sigma \rangle \rightsquigarrow \sigma'$, it is guaranteed that σ' satisfies Post*

Complexity: Assign unit cost to each application of a rule

Structured programs

Pros:

- Resembles modern high-level languages
- Allows for complex data types, can be extended with additional language features
- Easy to compose

Cons:

- Not as simple as Turing machines

Proving correctness

We will use an automated *proof assistant* to do proofs of programs

- You write the proof, the assistant checks it for you
- The ultimate TA; doesn't allow you to cheat

Dafny

`http://rise4fun.com/dafny`

State predicates

A **(state) predicate** is a boolean function on the program state.

- $x = 8$
- $x < y$
- $m \leq n \rightarrow (\forall j : 0 \leq j < a.length \cdot a[j] \neq NaN)$
- $true$
- $false$

Intuitively, a *property* that holds at a point in a program execution.

Hoare triples

For predicates P and Q and program S,

$$\{P\}S\{Q\}$$

says that if S is started at (a state satisfying) P, then it terminates at Q

Hoare triples

For predicates P and Q and program S,

$$\{P\}S\{Q\}$$

says that if S is started at (a state satisfying) P, then it terminates at Q

- P is a *precondition*
- Q is a *postcondition*

Examples

$$\{true\}x := 12\{x = 12\}$$
$$\{x < 40\}x := 12\{10 \leq x\}$$
$$\{x < 40\}x := x + 1\{x \leq 40\}$$
$$\{m \leq n\}j := (m + n)/2\{m \leq j \leq n\}$$

Examples

$$\{true\}x := 12\{x = 12\}$$
$$\{x < 40\}x := 12\{10 \leq x\}$$
$$\{x < 40\}x := x + 1\{x \leq 40\}$$
$$\{m \leq n\}j := (m + n)/2\{m \leq j \leq n\}$$
$$\{0 \leq m < n \leq a.length \wedge a[m] = x\}r := Find(a, m, n, x)\{m \leq r\}$$

Examples

$$\{true\}x := 12\{x = 12\}$$
$$\{x < 40\}x := 12\{10 \leq x\}$$
$$\{x < 40\}x := x + 1\{x \leq 40\}$$
$$\{m \leq n\}j := (m + n)/2\{m \leq j \leq n\}$$
$$\{0 \leq m < n \leq a.length \wedge a[m] = x\}r := Find(a, m, n, x)\{m \leq r\}$$
$$\{false\}S\{x^n + y^n = z^n\}$$

Precise triples

If $\{P\}S\{Q\}$ and $\{P\}S\{R\}$, then do we have

$$\{P\}S\{Q \wedge R\}?$$

Precise triples

If $\{P\}S\{Q\}$ and $\{P\}S\{R\}$, then do we have

$$\{P\}S\{Q \wedge R\}?$$

- Yes.

The most precise Q such that $\{P\}S\{Q\}$ is called the **strongest postcondition** of S with respect to P.

Weakest preconditions

If $\{P\}S\{R\}$ and $\{Q\}S\{R\}$, then $\{P \vee Q\}S\{R\}$ holds.

The most general P such that $\{P\}S\{R\}$ is called the **weakest precondition** of S with respect to R, written $wp(S, R)$.

Triples and wp

Here's the crucial relationship between Hoare triples and weakest preconditions:

$\{P\}S\{Q\}$ if and only if $P \rightarrow wp(S, Q)$

Proving programs correct

Consider programs of different shapes; for each shape, give method for systematic correctness proof

$$
\begin{array}{rcl}
e & ::= & \mathtt{f}(e_1, e_2) \mid x \mid \mathtt{n} \qquad \text{where } x \in \mathit{Var}, \mathtt{n} \in \mathit{Const}, \mathtt{f} \in \mathit{Func} \\
b & ::= & \mathtt{R}(e_1, e_2) \qquad \text{where } \mathtt{R} \in \mathit{Pred} \\
S & ::= & x := e \mid S_1; S_2 \mid \mathbf{skip} \mid \\
 & & \mathbf{if}\ b\ S_1\ \mathbf{else}\ S_2 \mid \mathbf{while}\ b\ S_1
\end{array}
$$

Proving programs correct: **skip**

$$wp(\textbf{skip}, R) \equiv R$$

Example:

$$wp(\textbf{skip}, x^n + y^n = z^n) \equiv x^n + y^n = z^n$$

To prove $\{P\}\textbf{skip}\{Q\}$, show that $P \rightarrow Q$

Proving programs correct: assignments

$$wp(w := E, R) \equiv R[w \mapsto E]$$

where $R[w \mapsto E]$ is obtained by starting with R and replacing w by E

$$wp(x := x + 1, x \leq 10) \equiv x + 1 \leq 10 \equiv x < 10$$
$$wp(x := 15, x \leq 10) \equiv 15 \leq 10 \equiv \mathit{false}$$
$$wp(y := x + 3 * y, x \leq 10) \equiv x \leq 10$$

To prove $\{P\} w := E \{Q\}$, show that $P \rightarrow Q[w \mapsto E]$

Dafny example

```
method foo(x: int) returns (y: int)
requires x > 0;
ensures y > 1;
{
   y := x + 1;
}
```

http://rise4fun.com/Dafny

http://research.microsoft.com/en-us/um/people/leino/papers/krml220.pdf

Sequential composition

$$wp(S; T, R) \equiv wp(S, wp(T, R))$$

Example: Compute the value of

$$wp(y := y + 1; x := x + 3 * y, y \leq 10 \wedge 3 \leq x)$$

Branching

$$
\begin{aligned}
& wp(\textbf{if } B \textbf{ then } S \textbf{ else } T, R) \\
\equiv \quad & (B \wedge wp(S, R)) \vee (\neg B \wedge wp(T, R))
\end{aligned}
$$

Branching

$$wp(\textbf{if } B \textbf{ then } S \textbf{ else } T, R) \equiv (B \wedge wp(S, R)) \vee (\neg B \wedge wp(T, R))$$

- $(B \wedge wp(S, R)$ is the set of states that pass the test "**if** B" *and* lead to R when "pushed" through S
- $(\neg B \wedge wp(S, R)$ is the set of states that fail the test "**if** B" *and* lead to R when "pushed" through T

Example: Compute the value of

- $wp(\textbf{if } x < y \textbf{ then } z := y \textbf{ else } z := x, 0 \leq z)$
- $wp(\textbf{if } x \neq 10 \textbf{ then } x := x + 1 \textbf{ else } x := x + 2, x \leq 10)$

Example

What's the condition under which this program will use pointers safely?

```
if (x != null) {
   n := x.f;
}
else {
   n := z-1;
   z := z + 1;
}
a = new char[n];
```

To prove

$$\{P\}\textbf{while } B\ S\{Q\}$$

find *invariant* J and a *ranking function* rf such that:

- invariant holds initially: $P \rightarrow J$
- invariant is maintained: $\{J \wedge B\}S\{J\}$
- invariant is sufficient: $J \wedge \neg B \rightarrow Q$
- ranking function is bounded: $J \wedge B \rightarrow 0 \leq rf$
- ranking function decreases: $\{J \wedge B \wedge rf = RF\}S\{rf < RF\}$

Example: Array sum

$\{N \geq 0\}$;

```
k := 0;
s := 0;
while (k != N) {
      s := s + a[k];
      k := k + 1
}
```

$\{s = \sum_{0 \leq i < N} a[i]\}$

Example: Array sum

$\{N \geq 0\}$

```
k := 0;
s := 0;
```

$\{J\}$

```
while (k != N) {
      s := s + a[k];
      k := k + 1
```

$\{J \wedge rf < RF\}$

```
}
```

$\{J \wedge \neg(k \neq N)\}$

$\{s = \sum_{0 \leq i < N} a[i]\}$

Example: Array sum

```
{N ≥ 0}
k := 0;
s := 0;
{J}
while (k != N) {
      s := s + a[k];
      k := k + 1
      {J ∧ rf < RF}
}
{J ∧ ¬(k ≠ N)}
{s = ∑_{0≤i<N} a[i]}
```

$$J : \quad s = \sum_{0\leq i<k} a[i] \wedge 0 \leq k \leq N$$
$$vf : \quad N - k$$

Exercise: Computing cubes

```
method Cube(N: int) returns (c: int)
  requires 0 <= N;
  ensures c == N*N*N;
{
  c := 0;
  var n := 0;
  var k := 1;
  var m := 6;
  while (n < N) {
    c := c + k;
    k := k + m;
    m := m + 6;
    n := n + 1;
  }
}
```

Exercise: Computing cubes

```
method Cube(N: int) returns (c: int)
  requires 0 <= N;
  ensures c == N*N*N;
{
  c := 0;
  var n := 0; var k := 1; var m := 6;
  while (n < N)
  invariant n <= N;
  invariant c == n*n*n;
  invariant k == 3*n*n + 3*n + 1;
  invariant m == 6*n + 6;
  decreases (N - n);
  {
    c := c + k;
    k := k + m;
    m := m + 6;
    n := n + 1;
  }
}
```

Question: Does this program terminate?

```
method Foo(x: int, y: int, z : int) {
  var x := x;
  var y := y;
  while (x > 0 && y > 0)
   {
    if (z == 1) {
      x := x - 1;
      y := y + 1;
    }
    else {
      y := y - 1;
    }
  }
}
```

Proving programs correct: Procedures

Suppose you have a program

```
method M( ){
  ...
  P( );
  ...
}
```

Proof goals:

- At the point when control enters `P` from within `M`, the ranking function of `P` must have a lower value than the ranking function of `M`
- *Assuming* a Hoare triple for `P`, *guarantee* a Hoare triple for `M`.

Proving programs correct: Procedures

```
method Ackermann(m: int, n: int) returns (r: int)
decreases m, n;
requires m >= 0 && n >= 0;
ensures r > 0;
{
  if (m <= 0) {
    r := n + 1;
  }
  else if (n <= 0) {
    r := Ackermann(m - 1, 1);
  }
  else {
    var z;
    z := Ackermann(m, n - 1);
    r := Ackermann(m - 1, z);
  }
}
```

Dafny also permits a “functional” notation:

```
function Ackermann(m: int, n: int): int
  decreases m, n;
  requires m >= 0 && n >= 0;
  ensures Ackermann(m,n) > 0;
{
  if m <= 0 then
    n + 1
  else if n <= 0 then
    Ackermann(m - 1, 1)
  else
    Ackermann(m - 1, Ackermann(m, n - 1))
}
```

Proving Fibonacci

Can you show that `Compute_Fib` is correct?

```
function Fib(n: nat): nat
{
  if n < 2 then n else Fib(n - 1) + Fib(n-2)
}

method Compute_Fib(n: nat) returns (x: nat)
ensures x == Fib(n);
{
  var i := 0;
  x := 0;
  var y := 1;
  while (i < n) {
   x, y := y, x + y;
   i := i + 1;
  }
}
```