Deductive Verification

Mooly Sagiv

Slides from Zvonimir Rakamaric
First-Order Logic

• A formal notation for mathematics, with expressions involving
  – Propositional symbols
  – Predicates
  – Functions and constant symbols
  – Quantifiers

• In contrast, propositional (Boolean) logic only involves propositional symbols and operators
First-Order Logic: Syntax

• As with propositional logic, expressions in first-order logic are made up of sequences of symbols
• Symbols are divided into *logical symbols* and *non-logical symbols* or *parameters*
• Example:

\[(x = y) \land (y = z) \land (f(z) \rightarrow f(x)+1)\]
First-Order Logic: Syntax

• Logical Symbols
  – Propositional connectives: $\land$, $\lor$, $\neg$, $\rightarrow$, ...
  – Variables: $V_1$, $V_2$, ...
  – Quantifiers: $\exists$, $\forall$

• Non-logical symbols/Parameters
  – Equality: $=$
  – Functions: $+$, $-$, $\%$, bit-wise $\&$, $f()$, concat, ...
  – Predicates: $\leq$, is_substring, ...
  – Constant symbols: 0, 1.0, null, ...
∀X:S. p(f(X),X) ⇒ ∃Y:S. p(f(g(X,Y)),g(X,Y)
Quantifier-free Subset

• We will largely restrict ourselves to formulas without quantifiers ($\forall, \exists$)
• This is called the quantifier-free subset/fragment of first-order logic with the relevant theory
Logical Theory

• Defines a set of parameters (non-logical symbols) and their meanings
• This definition is called a *signature*
• Example of a signature:
  Theory of linear arithmetic over integers
  Signature is \((0,1,+,\cdot,\cdot)\) interpreted over \(\mathbb{Z}\)
Presbruger Arithmetic

• Signature (0, 1, +, =) interpreted over \( \mathbb{Z} \)

• Axioms
  – \( \forall X: \mathbb{Z}. \neg ((X+1) = 0) \)
  – \( \forall X, Y: \mathbb{Z}. (X+1) = (Y+1) \Rightarrow X + Y \)
  – \( \forall X: \mathbb{Z}. X+0 = X \)
  – \( \forall X, Y: \mathbb{Z}. X+(Y+1) = (X+ Y)+1 \)
  – Let \( P(X) \) be a first order formula over \( X \)
    • \( (P(0) \land \forall X: \mathbb{Z}. P(X) \Rightarrow P(X+1)) \Rightarrow \forall Y: \mathbb{Z}. P(Y) \)
Many Sorted First Order Vocabulary

- A finite set of sorts $S$
- A finite set of function symbols $F$ each with a fixed signature $S^* \to S$
  - Zero arity functions are constant symbols
- A finite set of relation symbols $R$ each with a fixed arity $S^*$
An Interpretation $\mathfrak{I}$

- A **domain** $D_s$ for every $s \in S$
  
  - $D = \bigcup_{s \in S} D_s$

- For every function symbol $f \in F$, an interpretation $\mathfrak{I}[f]: D_{s_1} \times D_{s_2} \times \ldots \times D_{s_n} \rightarrow D_s$

- For every relation symbol $r \in R$, an interpretation $\mathfrak{I}[r] \subseteq D_{s_1} \times D_{s_2} \times \ldots \times D_{s_m}$
Many-Sorted First Order Formulas

• Logical Variables
  – Begin with Capital variables

• Typed Terms
  \(<\text{term}> ::= <\text{variable}> | f [(<\text{term}>, ... <\text{term}>)]\)

• Formulas
  \(<\text{form}> ::= <\text{term}> = <\text{term}> | r(<\text{term}>, ... <\text{term}>) // atomic\)
  \(<\text{form}> \lor <\text{form}> | <\text{form}> \land <\text{form}> | \neg <\text{form}> // Boolean\)
  \(\exists X: s<\text{form}> | \forall X : s. <\text{form}> // Quantifications\)
Free Variables

- FV: <term>, <formula> → 2^{Var}

- Terms
  - FV(X) = \{X\}
  - FV(f(t_1, t_2, ..., t_n)) = \bigcup_{i=1..n} FV(t_i)

- Formulas
  - FV(t_1 = t_2) = FV(t_1) \cup FV(t_2)
  - FV(r(t_1, t_2, ..., t_n)) = \bigcup_{i \in 1..n} FV(t_i)
  - FV(f_1 \lor, \land f_2) = FV(f_1) \cup FV(f_2)
  - FV(\neg f_2) = FV(f)
  - FV(\exists X:s.f) = FV(f) - \{X\}
  - FV(\forall X:s. f) = FV(f) - \{X\}
Assignments and Models

- **Assignment** $A$: $\text{Var} \rightarrow D$
- Extended to terms
  - $A(f(t_1, t_2, ..., t_n) = \iota[f](A(t_1), A(t_2), ..., A(t_n))$
- An assignment $A$ **models** a formula $f$ under interpretation $\iota$ (denoted by $A, \iota \models f$) if $f$ is true in $A$ (Tarsky’s semantics)
- $A, \iota \models t_1 = t_2$ if $A(t_1) = A(t_2)$
- $A, \iota \models r(t_1, t_2, ..., t_n)$ if $<A(t_1), A(t_2), ..., A(t_n)> \in \iota[r]$
- $A, \iota \models f_1 \lor f_2$ if $A, \iota \models f_1$ or $A, \iota \models f_2$
- $A, \iota \models \neg f$ if not $A, \iota \models f$
- $A, \iota \models \exists X: t. f$ if there exists $d \in D_t$ such that $A[X \mapsto d]$ if $A, \iota \models f$
A T-Interpretation

- A domain $D_s$ for every $s \in S$
  - $D = \bigcup_{s \in S} D_s$

- For every function symbol $f \in F$, an interpretation
  $\tau [f] : D_{s_1} \times D_{s_2} \times \ldots \times D_{s_n} \rightarrow D_s$

- For every relation symbol $r \in R$, an interpretation
  $\tau [r] \subseteq D_{s_1} \times D_{s_2} \times \ldots \times D_{s_m}$

- The domain and the interpretations satisfy the theory requirements(axioms)
Example Linear Arithmetic

• $S = \{\text{int}\}$, $F = \{0^0, 1^1, +^2\}$, $r = \{\leq^2\}$

• Domain
  – $D_{\text{int}} = \mathbb{Z}$

• Functions
  – $[0] = 0$
  – $[1] = 1$
  – $[+] = \lambda x, y: \text{int. } x + y$

• Relations
  – $[\leq] = \lambda x, y: \text{int. } x \leq y$
Assignments and T-Models

- **Assignment** $A$: $\text{Var} \rightarrow D$
- Extended to terms
  - $A(f(t_1, t_2, \ldots, t_n)) = \iota[f](A(t_1), A(t_2), \ldots, A(t_n))$
- An assignment $A$ which models a theory $T$, **T-models** a formula $f$ under interpretation $\iota$ (denoted by $A, \iota \models_T f$) if $f$ is true in $A$ (Tarsky’s semantics)
  - $A, \iota \models_T t_1 = t_2$ if $A(t_1) = A(t_2)$
  - $A, \iota \models_T r(t_1, t_2, \ldots, t_n)$ if $<A(t_1), A(t_2), \ldots, A(t_n)> \in \iota[r]$
  - $A, \iota \models_T f_1 \lor f_2$ if $A, \iota \models_T f_1$ or $A, \iota \models_T f_2$
  - $A, \iota \models_T \neg f$ if not $A, \iota \models_T f$
  - $A, \iota \models_T \exists X: t. f$ if there exists $d \in D_t$ such that $A[X \mapsto d]$ if $A, \iota \models_T f$
The SMT decision problem

- Input: A quantifier-free formula $f$ over a theory $T$
- Does there exist an $T$-interpretation $\tau$ and an assignment $A:FV(f) \rightarrow D$ such that $A \models_T f$
- The complexity depends on the complexity of the theory solvers
  - NPC-Undecidable
## Summary of Decidability Results

<table>
<thead>
<tr>
<th>Theory</th>
<th>Quantifiers Decidable</th>
<th>QFF Decidable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_E$</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>$T_{PA}$</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>$T_N$</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_Z$</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_R$</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_Q$</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$T_A$</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>
## Summary of Complexity Results

<table>
<thead>
<tr>
<th>Theory</th>
<th>Quantifiers</th>
<th>QF Conjunctive</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>NP-complete</td>
<td>O(n)</td>
</tr>
<tr>
<td>$T_E$ Equality</td>
<td>–</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>$T_N$ Presburger Arithmetic</td>
<td>O($2^2^2^2^2^2^2^2^2$)</td>
<td>NP-complete</td>
</tr>
<tr>
<td>$T_Z$ Linear Integer Arithmetic</td>
<td>O($2^2^2^2^2^2^2^2^2$)</td>
<td>NP-complete</td>
</tr>
<tr>
<td>$T_R$ Real Arithmetic</td>
<td>O($2^2^2^2^2^2^2^2^2$)</td>
<td>O($2^2^2^2^2^2^2^2^2$)</td>
</tr>
<tr>
<td>$T_Q$ Linear Rationals</td>
<td>O($2^2^2^2^2^2^2^2^2$)</td>
<td>PTIME</td>
</tr>
<tr>
<td>$T_A$ Arrays</td>
<td>–</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

$n$ – input formula size; $k$ – some positive integer
Basic Verifier Architecture

Program with specifications (assertions) → Verification condition generator → Verification condition (formula) → Theorem prover → Program correct or list of errors
Verification Condition Generator

• Creates verification conditions (mathematical logic formulas) from program’s source code
  – If VC is valid – program is correct
  – If VC is invalid – possible error in program

• Based on the theory of Hoare triples
  – Formalization of software semantics for verification

• Verification conditions computed automatically using weakest preconditions (wp)
Simple Command Language

\begin{align*}
&
x := E \\
&\text{havoc } x \\
&\text{assert } P \\
&\text{assume } P \\
&S ; T & \text{[sequential composition]} \\
&S \square T & \text{[choice statement]}
\end{align*}
Program States

• **Program state** $s$
  – Assignment of values (of proper type) to all program variables
  – Sometimes includes **program counter** variable $pc$
    • Holds current program location

• **Example**
  
  $s : (x \mapsto -1, y \mapsto 1)$
  $s : (pc \mapsto L, a \mapsto 0, i \mapsto 3)$

• **Reachable state** is a state that can be reached during some computation
Program States cont.

• A set of program states can be described using a FOL formula

• Example
  
  Set of states:
  
  \[ s : \{ (x \mapsto 1), (x \mapsto 2), (x \mapsto 3) \} \]

  FOL formulas defining \( s \):
  
  \[ x = 1 \land x = 2 \land x = 3 \]
  
  \[ 0 < x \land \exists x < 4 \quad [\text{if } x \text{ is integer}] \]
Hoare Triple

- Used for reasoning about (program) executions

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

- S is a command
- P is a **precondition** – formula about program state before S executes
- Q is a **postcondition** – formula about program state after S executes
Hoare Triple Definition

\{ P \} S \{ Q \}

- When a state $s$ satisfies precondition $P$, every terminating execution of command $S$ starting in $s$
  - does not go wrong, and
  - establishes postcondition $Q$
Hoare Triple Examples

- \{a = 2\} \ b := a + 3; \{b > 0\}
- \{a = 2\} \ b := a + 3; \{b = 5\}
- \{a > 3\} \ b := a + 3; \{a > 0\}
- \{a = 2\} \ b := a \times a; \{b > 0\}
Weakest Precondition [Dijkstra]

• The most general (i.e., weakest) P that satisfies

\[
\{ P \} S \{ Q \}
\]

is called the **weakest precondition** of S with respect to Q, written:

\[
wp(S, Q)
\]

• To check \( \{ P \} S \{ Q \} \) prove \( P \rightarrow wp(S, Q) \)
Weakest Precondition

• $wp: \text{Stm} \rightarrow (\text{Ass} \rightarrow \text{Ass})$
• $wp \lbrack S \rbrack(Q)$ – the weakest condition such that every terminating computation of $S$ results in a state satisfying $Q$
• $\sigma \models wp \lbrack S \rbrack(Q) \iff \forall \sigma': \sigma \lbrack S \rbrack \sigma' \rightarrow \sigma' \models Q$
Weakest Precondition [Dijkstra]

• The most general (i.e., weakest) $P$ that satisfies
  \[
  \{ P \} \mathcal{S} \{ Q \}
  \]
  is called the \textit{weakest precondition} of $S$ with respect to $Q$, written:
  \[
  \text{wp}(S, Q)
  \]

• To check $\{ P \} \mathcal{S} \{ Q \}$ prove $P \rightarrow \text{wp}(S, Q)$

• Example

  \[
  \{?P?\} \ b := a + 3; \ {b > 0}
  \]
  \[
  \{a + 3 > 0\} \ b := a + 3; \ {b > 0}
  \]
  \[
  \text{wp}(b := a + 3, \ b > 0) = a + 3 > 0
  \]
Strongest Postcondition

- The strongest Q that satisfies
  \[ \{ P \} S \{ Q \} \]
  is called the **strongest postcondition** of S with respect to P, written:
  \[ \text{sp}(S, P) \]
- To check \[ \{ P \} S \{ Q \} \] prove \[ \text{sp}(S, P) \rightarrow Q \]
- Strongest postcondition is (almost) a dual of weakest precondition
Weakest Preconditions Cookbook

• \( \text{wp}( x := E, \ Q ) = Q[ E / x ] \)
• \( \text{wp}( \text{havoc} \ x, \ Q ) = ( \forall x . \ Q ) \)
• \( \text{wp}( \text{assert} \ P, \ Q ) = P \land Q \)
• \( \text{wp}( \text{assume} \ P, \ Q ) = P \rightarrow Q \)
• \( \text{wp}( S ; T, \ Q ) = \text{wp}( S, \ \text{wp}( T, Q )) \)
• \( \text{wp}( S \Box T, \ Q ) = \text{wp}( S, Q) \land \text{wp}(T, Q) \)
Checking Correctness with wp

\{true\}

\x := 1;

\y := \x + 2;

assert \y = 3;
\{true\}
\{true\}
Checking Correctness with wp cont.

{true}

\( wp(x := 1, x + 2 = 3) = 1 + 2 = 3 \land true \)

\( x := 1; \)

\( wp(y := x + 2, y = 3) = x + 2 = 3 \land true \)

\( y := x + 2; \)

\( wp(\text{assert } y = 3, \text{ true}) = y = 3 \land true \)

\( \text{assert } y = 3; \)

{true}

Check: true \(\rightarrow\) 1 + 2 = 3 \land true
Example II

{x > 1}

\[ y := x + 2; \]

assert \( y > 3 \);

{true}
Example II cont.

\{x > 1\}

wp(y := x + 2, y > 3) = x + 2 > 3

y := x + 2;

wp(assert y > 3, true) = y > 3 \land true = y > 3

assert y > 3;

\{true\}

Check: x > 1 \rightarrow (x + 2 > 3)
Example III

{true}

assume x > 1;

y := x * 2;

z := x + 2;

assert y > z;

{true}
Example III cont.

{true}

wp(assume x > 1, x * 2 > x + 2) = x>1 → x*2 > x+2

assume x > 1;

wp(y := x * 2, y > x + 2) = x * 2 > x + 2

y := x * 2;

wp(z := x + 2, y > z) = y > x + 2

z := x + 2;

wp(assert y > z, true) = y > z ∧ true = y > z

assert y > z;

{true}
Structured if Statement

• Just a “syntactic sugar”:

```plaintext
if E then S else T
```
gets desugared into

```plaintext
(assume E ; S) □ (assume :E ; T)
```
Absolute Value Example

```c
if (x >= 0) {
    abs_x := x;
} else {
    abs_x := -x;
}
assert abs_x >= 0;
```
While Loop

while E
do
    S
end

• Loop body S executed as long as loop condition E holds
Desugar While Loop by Unrolling N Times

while E do S end =

if E {
    S;
    if E {
        S;
        if E {
            S;
            if E {assume false;} // blocks execution
        }
    }
}

Example

i := 0;
while i < 2 do i := i + 1 end

i := 0;
if i < 2 {
  i := i + 1;
  if i < 2 {
    i := i + 1;
    if i < 2 {
      i := i + 1;
      if i < 2 {assume false;} // blocks execution
    }
  }
}

}
First Issue with Unrolling

```
i := 0;
while i < 4 do i := i + 1 end

i := 0;
if i < 4 {
  i := i + 1;
  if i < 4 {
    i := i + 1;
    if i < 4 {
      i := i + 1;
      if i < 4 {assume false;} // blocks execution
    }
  }
}
```
Second Issue with Unrolling

\[
i := 0; \\
\text{while } i < n \text{ do } i := i + 1 \text{ end}
\]

\[
i := 0; \\
\text{if } i < n \{
    i := i + 1;
    \text{if } i < n \{
        i := i + 1;
        \text{if } i < n \{
            i := i + 1;
            \text{if } i < n \{\text{assume false;}\} // blocks execution
        \}
    \}
\}
\]
**While Loop with Invariant**

while \( E \)

\textbf{invariant} \( J \)

do

\( S \)

done

- **Loop body** \( S \) executed as long as \textit{loop condition} \( E \) holds

- **Loop invariant** \( J \) must hold on every iteration
  - \( J \) must hold initially and is evaluated before \( E \)
  - \( J \) must hold even on final iteration when \( E \) is false
  - \( J \) must be inductive
  - Provided by a user or inferred automatically
Desugaring While Loop Using Invariant

- while E invariant J do S end

assert J;

havoc x; assume J;

(assume E; S; assert J; assume false

assume ¬E

exit the loop

check that the loop invariant holds initially

jump to an arbitrary iteration of the loop

where x denotes the assignment targets of S

check that the loop invariant is maintained by the loop body
Weakest Precondition of While

- \( wp(\text{while } E \text{ invariant } J \text{ do } S \text{ end}, Q) = \)
Dafny

• Simple “verifying compiler”
  – Proves procedure contracts statically for all possible inputs
  – Uses theory of weakest preconditions

• Input
  – Annotated program written in simple imperative language
    • Preconditions
    • Postconditions
    • Loop invariants

• Output
  – Correct or list of failed annotations
Dafny Architecture

1. Program with specifications
2. Verification condition generator
3. Verification conditions
4. Theorem prover
5. Program correct or list of errors