CSCE-431: Verification

Jaakko Järvi

April 22, 2013
Outline

1 Introduction

2 Axiomatizing programming languages

3 Language specific rules

4 General rules

5 Termination

6 References

7 Dafny
Reliability

- **Correctness**
  - A system’s ability to perform according to its specification, in cases covered by the specification

- **Robustness**
  - A system’s ability to perform reasonably in cases not covered by its specification

- **Security**
  - A system’s ability to protect itself against hostile use
Establishing Correctness

- Testing, code reviews, informal reasoning, static analyses, etc. are all useful in strengthening our belief that a program is correct.
- If we wish to go further, to **guarantee correct behavior**, further techniques are necessary:
  - We need proofs.
- Two ways to obtain a proof that some requirement is satisfied—let this fact be expressed by formula $P$:
  1. By inspecting a program and the requirements, and through logical reasoning assure that $P$ must always be true. Often this takes place with a help of a **proof assistant**;
  2. or form a **semantic model** of the program, and check that $P$ holds in that model. This might mean “running” the model exhaustively to cover all possible states of the program. One might resort to using **model checkers**.
Goal for this course

- Basic understanding of static correctness proofs via theorem proving and model checking
Often the goal is not to prove the precise behavior of a program
Instead, the goal may be to prove some safety properties
  - the absence of references to deallocated memory or null-pointers
  - the absence of index-out-of-bounds errors
  - ...
Premises of correctness proofs

1. unambiguous language of specifying requirements
2. unambiguous language of specifying the meaning of implementations
Specifying implementations

- To reason about and to specify the meaning of a program, one first needs to specify the meanings of the constructs of a programming language
- Many approaches:
  - Operational semantics
    - Program is a stream of instructions to an *abstract machine*
    - The meaning of a program: behavior/result of the abstract machine
  - Denotational semantics
    - Define a meaning of each language construct in some suitable *semantic domain*
    - That is, define a *model* for the language
    - Meaning of a program arises, recursively, as a composition of the meanings of the sub-programs in the program
  - Axiomatic semantics
    - attach *proof rules* to each language construct, and eventually to each sub-program
    - proof rule: what is known to be true after the construct is execution, assuming a set of conditions were true prior to the execution
To reason about and to specify the meaning of a program, one first needs to specify the meanings of the constructs of a programming language.

Many approaches:

- **Operational semantics**
  - Program is a stream of instructions to an *abstract machine*.
  - The meaning of a program: behavior/result of the abstract machine.

- **Denotational semantics**
  - Define a meaning of each language construct in some suitable *semantic domain*.
  - That is, define a *model* for the language.
  - Meaning of a program arises, recursively, as a composition of the meanings of the sub-programs in the program.

- **Axiomatic semantics**
  - Attach *proof rules* to each language construct, and eventually to each sub-program.
  - Proof rule: what is known to be true after the construct is executed, assuming a set of conditions were true prior to the execution.
  - Seems to be most suitable for program verification, as well as for writing programs that are *correct by construction*.
Theory

- a mathematical framework for proving properties about a certain object domain

- Properties that are true are called *theorems*

- Components of a theory
  - Grammar (e.g., BNF)
    - defines *well-formed formulae* (WFF)
  - Axioms
    - formulae asserted to be theorems
  - *Inference rules*
    - ways to prove new theorems from previously obtained theorems

- Also: all formulae that can be proven true are a theory
What is a proof

Definition (Theorem)

A theorem $t$ in a theory is a well-formed formula of the theory, such that $t$ may be derived from the axioms by zero or more applications of the inference rules.
What is a proof

- The proof is a sequence of lines.
- Each line is numbered.
- Each line contains a formula, which the line asserts to be a theorem.
- Each line also contains a justification, an argument showing unambiguously that the formula of the line is indeed a theorem.
- The justification must be one of the following:
  - (A) the name of an axiom or axiom schema of the theory, in which case the formula must be the axiom or an instance of the axiom schema; or
  - (B) a list of references to previous lines, followed by a semicolon and the name of an inference rule or inference rule schema of the theory.
- In case B, the formulae on the lines referenced must coincide with the antecedents of the inference rule, and the formula on the current line must coincide with the consequent of the rule. (In the case of a rule schema, the coincidence must be with the antecedents and consequents of an instance of the rule.)
Discovering vs. checking a proof

- Discovering a proof requires insight
- Checking a proof can be mechanized
A theory is purely a formal/syntactic mechanism that defines a set of formulae, and allows for deriving some of the formulae as theorems.

What the formulae represent is not defined.

**Interpretation** of a theory:
- associate some member drawn from a suitable mathematical domain with every element in the theory’s vocabulary
- the association should be such that well-formed formulas are associated with Boolean values

**Model** of a theory:
- an interpretation that associates the value *true* with each theorem
1 Introduction

2 Axiomatizing programming languages

3 Language specific rules

4 General rules

5 Termination

6 References

7 Dafny
Theories about programs

- Formulas should be about properties of programs
- Properties can be expressed as assertions
- An assertion is a property of some objects in the program
  - it may or may not be true in a particular state of the program
- Usually the language’s concrete syntax of boolean expressions is used for assertions:
  \[
  a < b + 1
  \]
  \[
  true
  \]
  augmented with quantifiers and other math notation not part of the language
  \[
  \forall i \in \{1, \ldots, n\}. a[i] > a[i - 1]
  \]
- Conversely, some of the concrete syntax may not be allowed in assertions, such as functions with side effects
Assertions in programs

- **Floyd, 1967:**
  - use assertions as foundation for static correctness proofs
  - specify assertions at every program point
  - correctness reduced to reasoning about individual statements

- **Hoare, 1969:**
  - use Floyd’s ideas to define axiomatic semantics
  - that is, define the semantics of a programming language as a proof system

- **Dijkstra, 1975**
  - predicate transformer semantics
  - weakest preconditions
  - strongest postconditions
Well-formed formulas of an axiomatic theory for a programming language are not mere assertions, but expressions that consist of both assertions and program fragments.

Two kinds of assertions
- Preconditions, assumed to be satisfied before the fragment is executed.
- Postconditions, guaranteed to be satisfied after the fragment has been executed.

**Definition (Correct program)**

A program or program fragment is *correct* with respect to a certain precondition $P$ and a certain postcondition $Q$ if, when executed in a state in which $P$ is satisfied, it yields a state in which $Q$ is satisfied.
We can relate soundness and completeness to program executions:

- **Soundness**: every deduced property holds of all corresponding program executions
- **Completeness**: every property that holds of all program executions can be proven by the logic
  - This is of course undecidable
Well-formed formulas: “Hoare triples”

\{P\} A \{Q\}
Theory for expressing relevant properties of programs

Well-formed formulas: “Hoare triples”

\[ \{ P \} \ A \ \{ Q \} \]

- \( A \) is a program fragment
- \( P, Q \) predicates over the program state

*If \( A \) is started in any state that satisfies \( P \), the state after \( A \) terminates will satisfy \( Q \)*

- Hoare’s original notation was \( P \ \{ a \} \ Q \)
Total vs. partial correctness

\{P\} A \{Q\}

What if $A$ does not terminate?
What if $A$ does not terminate?

**Definition (Total correctness)**

A program fragment $A$ is *totally correct* with respect to $P$ and $Q$ if, when started in any state satisfying $P$, terminates in a state satisfying $Q$.

**Definition (Partial correctness)**

A program fragment $A$ is *partially correct* with respect to $P$ and $Q$ if, when started in any state satisfying $P$, if it terminates, does so in a state satisfying $Q$.
Partial correctness

@pre: {}
while (true) {
    // lots of ingenious work here
}
@post: { P == NP }
A program that does not terminate is partially correct wrt. to any specification.

Regardless, partial correctness often used, because proving termination tends to require different proof-techniques.

Concerns about termination separated from the concerns about correctness, and assumed to be established through other means.
Examples

\{\top\} \ x = 5 \ \{x = 5\}
Examples

\{\top\} \ x = 5 \ \{x = 5\}

\{x = y\} \ x = x + 3 \ \{x = y + 3\}
Examples

\{\top\} x = 5 \{x = 5\}

\{x = y\} x = x + 3 \{x = y + 3\}

\{x > 0\} x = x \ast 2 \{x > -2\}
Examples

\[
\{\top\} \ x = 5 \ \{x = 5\}
\]

\[
\{x = y\} \ x = x + 3 \ \{x = y + 3\}
\]

\[
\{x > 0\} \ x = x \times 2 \ \{x > -2\}
\]

\[
\{x = a\} \text{ if } (x < 0) \text{ then } x = -x \ \{x = |a|\}
\]
Examples

{\top} \ x = 5 \ \{x = 5\}

{x = y} \ x = x + 3 \ \{x = y + 3\}

{x > 0} \ x = x \times 2 \ \{x > -2\}

{x = a} \ \text{if} \ (x < 0) \ \text{then} \ x = -x \ \{x = |a|\}

{\bot} \ x = 3 \ \{x = 8\}
Outline

1. Introduction
2. Axiomatizing programming languages
3. Language specific rules
4. General rules
5. Termination
6. References
7. Dafny
To give an axiomatic semantics for a programming language, we define an inference rule for each different kind of statement.
A small language

\[ S ::= \]
\[ \quad V \leftarrow E \]
\[ \quad | \text{if } B \text{ then } S \text{ else } S \]
\[ \quad | \text{while } B \text{ do } S \]
\[ \quad | S ; S \]
\[ \quad | \text{skip} \]

\[ E ::= \text{expression} \]

\[ V ::= \text{variable name} \]

\[ B ::= \text{Boolean expression} \]
⊢ \{ P \} \text{skip} \{ P \}
⊢ {[e/x]Q} \ x \leftarrow \ e \ \{Q\}

- \([e/x]Q\) is the expression obtained from \(Q\) by substituting \(e\) for \(x\) in every (free) occurrence of \(x\).
- Intuitively, the rule says:
  - Whatever is true of \(x\) after assigning \(e\) to \(x\), must have been true of \(e\) before.
- The rule can be viewed as a function from a state and a postcondition to a precondition.
  - \(wp(S, Q)\)
Examples

$\vdash \{ y > 0 \} \ x \leftarrow y \{ x > 0 \}$
Examples

⊢ \{ y > 0 \} x \leftarrow y \{ x > 0 \}

⊢ \{ x + 1 > 0 \} x \leftarrow x + 1 \{ x > 0 \}
Examples

- $\vdash \{y > 0\} \ x \leftarrow y \{x > 0\}$
- $\vdash \{x + 1 > 0\} \ x \leftarrow x + 1 \ {\{x > 0\}}$
- $\vdash \{1 + 2 = 3\} \ x \leftarrow x + 1 \ {\{1 + 2 = 3\}}$
- $\vdash \{2 = 2\} \ x \leftarrow 2 \ {\{x = 2\}}$
Examples

- $\vdash \{y > 0\} \ x \leftarrow y \{x > 0\}$
- $\vdash \{x + 1 > 0\} \ x \leftarrow x + 1 \ \{x > 0\}$
- $\vdash \{1 + 2 = 3\} \ x \leftarrow x + 1 \ \{1 + 2 = 3\}$
- $\vdash \{2 = 2\} \ x \leftarrow 2 \ \{x = 2\}$
int hiding_trouble (int& x) {
  ++x;
  ++global;
  return 0;
}

Now consider these?

{global == 0} x = hiding_trouble (x) {global == 0}
{x == 0} y = hiding_trouble (x) {x == 0}
\[
\{B \land P\} \quad S_1 \quad \{Q\} \quad \{\neg B \land P\} \quad S_2 \quad \{Q\}
\]

\[
\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ end } \{Q\}
\]
\[
\{P\} S_1 \{Q\} \quad \{Q\} S_2 \{R\}
\]

\[
\{P\} S_1 ; S_2 \{R\}
\]
Loops

- To define the rule for loops, we need the notion of a *loop invariant*

**Definition (Loop Invariant)**

A predicate that is true when entering in a loop, when entering a new iteration of a loop, and immediately after exiting the loop is a *loop invariant*.

**Example: shopping for groceries**

```plaintext
cart = empty;
{ groceries wanted = groceries unchecked + groceries in cart}
while (grocery list not empty) {
    { groceries wanted = groceries unchecked + groceries in cart and
        list not empty }
    add grocery item to cart;
    check grocery off the list;
    { groceries wanted = groceries unchecked + groceries in cart }
}
{ groceries wanted = groceries unchecked + groceries in cart}
```
Loops

To define the rule for loops, we need the notion of a *loop invariant*.

**Definition (Loop Invariant)**

A predicate that is true when entering in a loop, when entering a new iteration of a loop, and immediately after exiting the loop is a *loop invariant*.

**Example: shopping for groceries**

```plaintext
cart = empty;
{ groceries wanted = groceries unchecked + groceries in cart}
while (grocery list not empty) {
    { groceries wanted = groceries unchecked + groceries in cart and list not empty }
    add grocery item to cart;
    check grocery off the list;
    { groceries wanted = groceries unchecked + groceries in cart }
}
{ groceries wanted = groceries unchecked + groceries in cart}
groceries wanted = groceries unchecked + groceries in cart
```
\[
\{ B \land I \} \ S \ \{ I \}
\]

\[
\{ I \} \text{ while } B \text{ do } S \text{ end } \{ \neg B \land I \}
\]
Outline

1. Introduction
2. Axiomatizing programming languages
3. Language specific rules
4. General rules
5. Termination
6. References
7. Dafny
Rule of consequence

\[
\begin{align*}
\text{Consequence} \\
\{ P \} \ a \ \{ Q \} & \quad P' \implies P & \quad Q \implies Q' \\
\{ P' \} & \ a \ \{ Q' \}
\end{align*}
\]
Rule of consequence

\[
\begin{array}{c}
\text{Consequence} \\
\{P\} \quad a \quad \{Q\} \quad \quad P' \quad \implies \quad P \quad \quad Q \quad \implies \quad Q' \\
\hline
\{P'\} \quad a \quad \{Q'\}
\end{array}
\]

- Example: assume \(x\) and \(y\) are integers and that we can prove:

\[
\vdash \{x + x > 2\} \quad y = x + x \quad \{y > 1\}
\]

- We, however, we wish to obtain:

\[
\vdash \{x > 1\} \quad y = x + x \quad \{y > 1\}
\]
Example: assume $x$ and $y$ are integers and that we can prove:

$$\vdash \{x + x > 2\} \ y = x + x \ \{y > 1\}$$

We, however, we wish to obtain:

$$\vdash \{x > 1\} \ y = x + x \ \{y > 1\}$$

We can apply the rule of consequence:

$$\begin{align*}
\{x + x > 2\} \ y = x + x \ \{y > 1\} & \quad \text{and} \quad \{x > 1\} \Rightarrow \{x + x > 2\} \\
\{y > 1\} \Rightarrow \{y > 1\}
\end{align*}$$
What is the **full** theory?

- Consider \( \{ x > 1 \} \implies \{ x + x > 2 \} \)
- We seem to rely on our basic understanding of mathematics here.
- Can we do so? What do the symbols \( > \) and \( + \) refer to?
What is the **full** theory?

- Consider \( \{x > 1\} \implies \{x + x > 2\} \)

- We seem to rely on our basic understanding of mathematics here.

- Can we do so? What do the symbols \( > \) and \( + \) refer to?

- The expressions in the assertions are from the host programming language (e.g., Java or C++)
  - the mathematical denotations of \( > \) and \( + \) should not be assumed
  - note that, e.g., \( \text{int} \) and \( \mathbb{Z} \) are quite different

- To statically check proofs, such a theory of “elementary mathematics” (and elementary logic) need to be developed and integrated with the theory of assertions and pre- and post-conditions for a particular language.
What is the full theory?

Consider \( \{x > 1\} \implies \{x + x > 2\} \)

We seem to rely on our basic understanding of mathematics here.

Can we do so? What do the symbols \( > \) and \( + \) refer to?

The expressions in the assertions are from the host programming language (e.g., Java or C++)

- the mathematical denotations of \( > \) and \( + \) should not be assumed
- note that, e.g., \texttt{int} and \( \mathbb{Z} \) are quite different

To statically check proofs, such a theory of “elementary mathematics” (and elementary logic) need to be developed and integrated with the theory of assertions and pre- and post-conditions for a particular language

We now assume that such a theory of has been defined, and whenever relying on it, mark it as EM in proofs
Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>[\frac{x + x &gt; 2}{y = x + x {y &gt; 1}}]</td>
<td>Assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>[{x &gt; 1} \implies {x + x &gt; 2}]</td>
<td>EM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>[{y &gt; 1} \implies {y &gt; 1}]</td>
<td>EM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>[{x &gt; 1} y = x + x {y &gt; 1}]</td>
<td>T1, T2, T3; Consequence</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In practice, developing theories for particular object domains, such as the theory of EM, can be difficult. Consider, e.g., floating point numbers. Such theories, however, are reusable.
Example

| T1  | \( \vdash \{ x + x > 2 \} \ y = x + x \ \{ y > 1 \} \) | Assignment |
| T2  | \( \{ x > 1 \} \implies \{ x + x > 2 \} \) | EM |
| T3  | \( \{ y > 1 \} \implies \{ y > 1 \} \) | EM |
| T4  | \( \{ x > 1 \} \ y = x + x \ \{ y > 1 \} \) | T1, T2, T3; Consequence |

- In practice, developing theories for particular object domains, such as theory of EM, can be difficult
  - Consider, e.g., floating point numbers
- Such theories, however, are reusable
Rule of conjunction

Conjunction

\[
\{P\} \ a \ \{Q\} \quad \{P\} \ a \ \{R\}
\]

\[
\{P\} \ a \ \{Q \land R\}
\]
Termination proofs

- The problematic language constructs are loops (or recursion, but we do not have it in our small language)
- Basic idea of termination proofs is that if one can show that during each loop iteration, some measure gets smaller and eventually reaches a value from which it can no longer decrease, and the loop cannot go on forever
Termination proofs

- The problematic language constructs are loops (or recursion, but we do not have it in our small language)

- Basic idea of termination proofs is that if one can show that during each loop iteration, some measure gets smaller and eventually reaches a value from which it can no longer decrease, and the loop cannot go on forever

- Must get substantially smaller
  - Like this: 10, 9, 8, 5, 2, ...
  - Not like this: 1, 1/2, 1/4, 1/8, 1/16, ...
Loop variant formally

- Rule for partial correctness

\[
\begin{align*}
\{ B \land I \} & \rightarrow S \{ I \} \\
\{ I \} & \text{ while } B \text{ do } S \text{ end } \{ \neg B \land I \}
\end{align*}
\]
Loop variant formally

- Rule for partial correctness

\[
\{B \land I\} \ S \ \{I\} \\
\{I\} \ \text{while } B \ \text{do } S \ \text{end} \ \{\neg B \land I\}
\]

- Rule for total correctness

  - \(V\), expression of type \texttt{int}, is a \textit{variant}
  - \(z\) is fresh, not written to in \(B\)

\[
\{(B \land I) \implies V > 0\} \ \{B \land I\} \ z \leftarrow V; S \ \{I \land V < z\} \\
\{I\} \ \text{while } B \ \text{do } S \ \text{end} \ \{\neg B \land I\}
\]
Outline

1 Introduction
2 Axiomatizing programming languages
3 Language specific rules
4 General rules
5 Termination
6 References
7 Dafny
References

- *An Axiomatic Basis for Computer Programming*, C.A.R. Hoare, CACM 12(10), 1969
Dafny

- Language designed to be easy for writing provably correct code
- Correct means
  - no runtime errors (index out of bounds, null dereferences, etc.)
  - correct wrt. to the intended meaning, where specification given in annotations
Function definitions equipped with annotations
  - preconditions
  - postconditions
  - assertions
  - loop invariants and variants

For each function/method, Dafny searches for a proof that the implementation satisfies the annotations for all inputs

Dafny also proofs termination, unless it is explicitly asked not to for some loops
Two reasons why a program is rejected are

1. Dafny finds annotations inconsistent with the implementation
2. Dafny is not “clever” enough to prove that the annotations are consistent with the implementation

The former case indicates a bug in the code (or in annotations), the latter that programmer must help Dafny with more annotations
About Dafny

- Typical imperative language
- From Rustan Leino et al., Microsoft Research
- To experiment without installing
  - http://rise4fun.com/dafny/Hello