Outline

1. Introduction

2. Axiomatizing programming languages

3. Language specific rules

4. Termination

5. Calculus of weakest preconditions

6. References
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
What properties to prove

- 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
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.
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.
    - 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
Example theory

Grammar

\[ \gamma, \delta ::= \alpha_1 = \alpha_2 \]

\[ \alpha_1 < \alpha_2 \]

\[ \neg \gamma \]

\[ \gamma \implies \delta \]

\[ \alpha ::= 0 \mid n \mid \alpha' \]

*\( n \) is a *metavariable* ranging over variable names

*\( \gamma, \delta \) range over boolean expressions

*\( \alpha \) ranges over integer expressions

*\( (\cdot)' \) is the successor operator

Which of these are WFFs?

- \( 0 = 0 \)
- \( 0 \neq 0 \)
- \( 0 < 1 \)
- \( 0 = 0 \implies 0' = 0' \)
- \( m''' < 0'' \)
Example theory continues

- $f$ is a theorem is expressed as

\[ \vdash f \]

- axioms and inference rules, jointly “rules”, define which of the well-formed formulae are theorems
- no need to care about expressions that are not WFF
The latter is really a rule schema, for it mentions some metavariables ($n_1$, $n_2$, $n'_1$, and $n'_2$) substituting all metavariables for integer expressions will give a rule.

For example:

$\vdash 0 < 0'$

$\vdash n_1 < n_2 \implies n'_1 < n'_2$
Axioms

\[ A0 \quad \vdash 0 < 0' \]

\[ AS \quad \vdash n_1 < n_2 \implies n_1' < n_2' \]

- The latter is really a *rule schema*, for it mentions some *metavariables* \((n_1, n_2, n_1', n_2')\)
- substituting all metavariables for integer expressions will give a *rule*
- For example:

\[ \vdash 0' < 0'''' \implies 0'' < 0''''' \]
Inference rules

- **MP**
  \[ \gamma \quad \gamma \implies \delta \]
  \[ \delta \]

- **Ind**
  \[ \gamma(0) \quad \gamma(n) \implies \gamma(n') \]
  \[ \gamma(n) \]

- If the formulae above the bar (*antecedents*) are theorems, then the formula below the bar (*consequent*) is a theorem.

- \( \gamma(x) \) is syntax for some formula \( \gamma \) that has one or more occurrences of a *parameter*, and \( x \) is substituted for all occurrences of that parameter.
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 is preceded by an implicit $\vdash$).
- 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.)
Prove that $n < n'$

<table>
<thead>
<tr>
<th>Number</th>
<th>Formula</th>
<th>Justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>M.1</td>
<td>$0 &lt; 0'$</td>
<td>A0</td>
</tr>
<tr>
<td>M.2</td>
<td>$n &lt; n' \implies n' &lt; n''$</td>
<td>AS</td>
</tr>
<tr>
<td>M.3.</td>
<td>$n &lt; n'$</td>
<td>M.1, M.2; Ind</td>
</tr>
</tbody>
</table>
Discovering vs. checking a proof

- Discovering a proof requires insight
- Checking a proof can be made mechanical
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.
Example models

- Integers
  - associate the integer 0 with the symbol 0
  - associate the successor operation on integers to \((\cdot)\)'
  - associate the integer equality with the symbol \(=\)
  - etc.
Example models

- **Integers**
  - associate the integer 0 with the symbol 0
  - associate the successor operation on integers to \( \cdot \)'
  - associate the integer equality with the symbol =
  - etc.

- **Persons**
  - 0 is interpreted as some person
  - \( x' \) is the mother of \( x \)
  - \( x < y \) is true if \( y \) is the maternal ancestor of \( x \)
  - etc.
Soundness and completeness

- Soundness: a theory is sound if for no well-formed formula $f$ the rules allow deriving both $f$ and $\neg f$.
- Completeness: a theory is complete if for any well-formed formula $f$ the rules allow the derivation of $f$ or $\neg f$. 

There are useful theories that are not complete.
Soundness and completeness

- **Soundness**: a theory is sound if for no well-formed formula $f$ the rules allow deriving both $f$ and $\neg f$.
- **Completeness**: a theory is complete if for any well-formed formula $f$ the rules allow the derivation of $f$ or $\neg f$

- A theory is sound, if and only if it has a model
- There are useful theories that are not complete
For a theory $A$, a model $M$ of $A$ and a property $p$:

- $M \models p$
  - means that $p$ can be proved from $M$

- $A \vdash p$
  - means that $p$ can be proved from $A$
For a theory $A$, a model $M$ of $A$ and a property $p$:

- $M \models p$
  - means that $p$ can be proved from $M$

- $A \vdash p$
  - means that $p$ can be proved from $A$

- Soundness: $A \vdash p \implies M \models p$
- Completeness: $M \models p \implies A \vdash p$
1. Introduction

2. Axiomatizing programming languages

3. Language specific rules

4. Termination

5. Calculus of weakest preconditions

6. References
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
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.
Soundness and completeness again

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 proved by the logic
  - This is of course undecidable
Well-formed formulas: “Hoare triples”

\[ \{ P \} \ A \ \{ Q \} \]
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 \( P \ \{a\} \ Q \)
Your boss tells you to do $A$, and specifies the assignment as

$\{P\} \ A \ \{Q\}$

Would you prefer a weak or strong $P$ (respectively $Q$)?

- If $P \implies Q$ and $\neg(Q \implies P)$, then $P$ is stronger than $Q$
Your boss tells you to do \( A \), and specifies the assignment as \( \{ P \} \ A \ \{ Q \} \).

Would you prefer a \textit{weak} or \textit{strong} \( P \) (respectively \( Q \))?

- If \( P \implies Q \) and \( \neg(Q \implies P) \), then \( P \) is stronger than \( Q \) (and \( Q \) weaker than \( P \)).
Your boss tells you to do $A$, and specifies the assignment as $\{P\} \ A \ \{Q\}$

Would you prefer a weak or strong $P$ (respectively $Q$)?

If $P \implies Q$ and $\neg(Q \implies P)$, then $P$ is stronger than $Q$ (and $Q$ weaker than $P$)

Consider

\[
\begin{align*}
\{\bot\} & \ A \ \{\ldots\} \\
\{\ldots\} & \ A \ \{\top\} \\
\{\top\} & \ A \ \{Q\} \\
\{P\} & \ A \ \{\bot\}
\end{align*}
\]
Your boss tells you to do \( A \), and specifies the assignment as \( \{ P \} \ A \ \{ Q \} \).

Would you prefer a weak or strong \( P \) (respectively \( Q \))?

- If \( P \implies Q \) and \( \neg (Q \implies P) \), then \( P \) is stronger than \( Q \) (and \( Q \) weaker than \( P \)).

Consider

\[
\begin{align*}
\{ \bot \} & \ A \ \{ \ldots \} \\
\{ \ldots \} & \ A \ \{ \top \} \\
\{ \top \} & \ A \ \{ Q \} \\
\{ P \} & \ A \ \{ \bot \}
\end{align*}
\]

- If \( P \) is as strong as possible, and \( Q \) as weak as possible, then the job of \( A \) is the easiest.
Assertion? Two views

- A function from the set of states to Booleans
  - $\top$: maps all states to true
  - $\bot$: maps all states to false
  - $P \implies Q$: $\forall s : \text{State}, P(s) \implies Q(s)$
  - etc. for other logical connectives

- A subset of all states (the set where assertion satisfied)
  - $\top$: the entire set
  - $\bot$: the empty set
  - $P \implies Q$: $P \subset Q$
  - $P \land Q$: $P \cap Q$
  - $P \lor Q$: $P \cup Q$
Total vs. partial correctness

\{P\} A \{Q\}

- What if \( A \) does not terminate?
Total vs. partial correctness

\{ P \} A \{ Q \}

What if \( A \) does not terminate?

Definition (Total correctness)

A program fragment \( A \) is \textit{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 \textit{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 \( Q \).
@pre: {}

while (true) {
    // lots of ingenious work here
}

@post: { P == NP }
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 \times 2 \ \{x > -2\}
Examples

{⊤} 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 \ast 2 \ \{x > -2\}

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

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

\[
\begin{align*}
\{P\} & \quad a \quad \{Q\} \\
\implies P' & \quad \implies \quad P & \quad Q & \implies \quad Q' \\
\{P'\} & \quad a \quad \{Q'\}
\end{align*}
\]
General inference rules

**Consequence**

\[
\begin{array}{c}
\{P\} \ a \ \{Q\} \\
\hline
P' \implies P & Q \implies Q' \\
\{P'\} \ a \ \{Q'\}
\end{array}
\]

- 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:
  \[\vdash \{x > 1\} \ y = x + x \ \{y > 1\}\]
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:

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

$$\{x > 1\} \quad \Rightarrow \quad \{x + x > 2\} \quad \{y > 1\} \quad \Rightarrow \quad \{y > 1\}$$

$$\{x > 1\} \ y = x + x \ \{y > 1\}$$
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
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td>$\vdash { x + x &gt; 2 } \ y = x + x \ { y &gt; 1 }$ (proved separately)</td>
<td></td>
</tr>
<tr>
<td><strong>T2</strong></td>
<td>${ x &gt; 1 } \implies { x + x &gt; 2 }$</td>
<td>EM</td>
</tr>
<tr>
<td><strong>T3</strong></td>
<td>${ y &gt; 1 } \implies { y &gt; 1 }$</td>
<td>EM</td>
</tr>
<tr>
<td><strong>T4</strong></td>
<td>${ x &gt; 1 } \ y = x + x \ { y &gt; 1 }$</td>
<td>T1, T2, T3; Consequence</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Theory</th>
<th>Premise</th>
<th>Conclusion</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>${x + x &gt; 2}, y = x + x {y &gt; 1}$</td>
<td>(proved separately)</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>${x &gt; 1}$</td>
<td>$\implies {x + x &gt; 2}$</td>
<td>EM</td>
</tr>
<tr>
<td>T3</td>
<td>${y &gt; 1}$</td>
<td>$\implies {y &gt; 1}$</td>
<td>EM</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>
</tr>
</tbody>
</table>

- 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
What about deriving \( \vdash \{ P \} \ a \ \{ Q \} \) or \( \vdash \{ P \} \ a \ \{ R \} \) from \( \vdash \{ P \} \ a \ \{ Q \ \land \ R \} \)?

Follow from EM theorems:

\[(Q \land R) = \Rightarrow Q \text{ and } (Q \land R) = \Rightarrow R\]

and the rule of consequence.
Conjunction

\[
\begin{array}{c}
\{P\} \vdash \{Q\} \\
\{P\} \vdash \{R\}
\end{array}
\]

\[
\{P\} \vdash \{Q \land R\}
\]

What about deriving \( \vdash \{P\} \vdash \{Q\} \) or \( \vdash \{P\} \vdash \{R\} \) from \( \vdash \{P\} \vdash \{Q \land R\}\)?
Rule of conjunction

Conjunction

\[
\begin{array}{c}
\{P\} \vdash \{Q\} \\
\{P\} \vdash \{R\}
\end{array}
\]

\[
\{P\} \vdash \{Q \land R\}
\]

What about deriving \(\vdash \{P\} \vdash \{Q\}\) or \(\vdash \{P\} \vdash \{R\}\) from \(\vdash \{P\} \vdash \{Q \land R\}\)\

Follow from EM theorems \((Q \land R) \implies Q\) and \((Q \land R) \implies R\) and the rule of consequence.
1. Introduction
2. Axiomatizing programming languages
3. Language specific rules
4. Termination
5. Calculus of weakest preconditions
6. References
To give an axiomatic semantics for a programming language, we define an inference rule for each different kind of statement.
\[ \vdash \{ P \} \text{skip} \{ P \} \]
Abort

\[ \vdash \{ \bot \} \text{abort} \{ Q \} \]
⊢ $\{\text{[}e/\text{x}\text{]}Q\} \ x \leftarrow \ e \ \{Q\}$

- $\text{[}e/\text{x}\text{]}Q$ 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

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

\[ \vdash \{ y > 0 \} \ x \leftarrow y \{ x > 0 \} \]
\[ \vdash \{ x + 1 > 0 \} \ x \leftarrow x + 1 \{ x > 0 \} \]
\[ \vdash \{ 1 + 2 = 3 \} \ x \leftarrow x + 1 \{ 1 + 2 = 3 \} \]
\[ \vdash \{ 2 = 2 \} \ x \leftarrow 2 \{ x = 2 \} \]

42 / 66
Side-effects?

```c
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}
\[ \{ B \land P \} \ S_1 \ \{ Q \} \quad \{ \neg B \land P \} \ S_2 \ \{ Q \} \]

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

$\{P\} \ S_1; \ S_2 \ \{R\}$
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}
```
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
```
\begin{align*}
\{B \land I\} & \quad S \quad \{I\} \\
\{I\} & \quad \text{while } B \quad \text{do } S \quad \text{end } \quad \{\neg B \land I\}
\end{align*}
1. Introduction
2. Axiomatizing programming languages
3. Language specific rules
4. Termination
5. Calculus of weakest preconditions
6. References
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 substantially 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

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

- Rule for partial correctness

\[
\begin{align*}
\{B \land I\} & \quad S \quad \{I\} \\
\{I\} & \quad \text{while } B \text{ do } S \text{ end } \{\neg B \land I\}
\end{align*}
\]

- Rule for total correctness

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

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

1. Introduction
2. Axiomatizing programming languages
3. Language specific rules
4. Termination
5. Calculus of weakest preconditions
6. References
Pre-post semantics

- What we have seen thus far, flexibility in specifying pre and post conditions
- From

\[ \vdash \{ P \} \ a \ \{ Q \} \]

we get

\[ \vdash \{ P' \} \ a \ \{ Q' \} \]

for any \( P' \) stronger than \( P \) and for any \( Q' \) weaker than \( Q \)

- The former formula is more “interesting” or “informative”
Pre-post semantics

- What we have seen thus far, flexibility in specifying pre and post conditions
- From

\[ \vdash \{ P \} a \{ Q \} \]

we get

\[ \vdash \{ P' \} a \{ Q' \} \]

for any \( P' \) stronger than \( P \) and for any \( Q' \) weaker than \( Q \)

- The former formula is more “interesting” or “informative”
- Often the “most interesting” formula is not the most interesting formula when proving specific properties of programs
What we have seen thus far, flexibility in specifying pre and post conditions.

From $\vdash \{P\} a \{Q\}$

we get

$\vdash \{P'\} a \{Q'\}$

for any $P'$ stronger than $P$ and for any $Q'$ weaker than $Q$.

The former formula is more “interesting” or “informative”.

Often the “most interesting” formula is not the most interesting formula when proving specific properties of programs — but when defining the semantics of a language, it is...
Inferring the most interesting specifications

- All the inference rules we saw, except for loops, are the most interesting statements about each language construct, in the sense that
  - given a postcondition $Q$ for a statement $S$, $P$ is the weakest possible precondition, and
  - given a precondition $P$ for a statement $S$, $Q$ is the strongest possible postcondition.

- In general, we cannot infer the most interesting pair $P, Q$ for some $S$

- To restrict the arbitrary choice, we can fix either $P$ or $Q$ and infer the missing condition such that we get the most interesting pair
  - $S \ wp \ Q$
  - $S \ sp \ P$
Definition (Weakest Precondition)

The wp-formula $S \text{ wp } Q$, where $S$ is a statement and $Q$ an assertion, denotes the weakest assertion $P$ such that $S$ is totally correct with respect to precondition $P$ and post condition $Q$.

- The goal is to provide a calculus, with mechanical transformation rules
- Impractical goal (loops are the culprit), but still a useful theory
- Next: brief look at rewrites of the pre-post inference rules in wp-calculus
What about $\vdash S \text{ wp } \bot = \bot$? Since any state satisfies $\top$, $S \text{ wp } \top$ is the weakest precondition that guarantees that $S$ terminates.
True and False

\[ \vdash S \text{ wp } \bot = \bot \]

What about

\[ \vdash S \text{ wp } \top = ? \]
True and False

\[ \vdash S \text{ wp } \bot = \bot \]

- What about \[ \vdash S \text{ wp } \top = ? \]

- Since any state satisfies \( \top \), \( S \text{ wp } \top \) is the weakest precondition that guarantees that \( S \) terminates
Rule of consequence

WP-Consequence

\[
Q \implies Q' \\
(S \ wp \ Q) \implies (S \ wp \ Q')
\]
Rule of consequence

WP-Consequence

\[ Q \implies Q' \]

\[ (S \text{ wp } Q) \implies (S \text{ wp } Q') \]

Since \( Q \) is stronger than \( Q' \), any initial condition that guarantees \( a \) terminates satisfying \( Q \), also guarantees that \( a \) terminates satisfying \( Q' \).
Rule of conjunction

WP-Conjunction

\[(S \text{ wp } Q \land R) = (S \text{ wp } Q) \land (S \text{ wp } R)\]

WP-Disjunction

\[(S \text{ wp } Q \lor R) = (S \text{ wp } Q) \lor (S \text{ wp } R)\]
Rule of conjunction

WP-Conjunction

\((S \text{ wp } Q \land R) = (S \text{ wp } Q) \land (S \text{ wp } R)\)

WP-Disjunction

\((S \text{ wp } Q \lor R) = (S \text{ wp } Q) \lor (S \text{ wp } R)\)

Does the disjunction rule hold?
WP-Conjunction

\((S \text{ wp } Q \land R) = (S \text{ wp } Q) \land (S \text{ wp } R)\)

WP-Disjunction

\((S \text{ wp } Q \lor R) = (S \text{ wp } Q) \lor (S \text{ wp } R)\)

Does the disjunction rule hold?

\(\iff \text{ yes}\)

\(\implies \text{ ???}\)
Rule of conjunction

WP-Conjunction
\[(S \text{ wp } Q \land R) = (S \text{ wp } Q) \land (S \text{ wp } R)\]

WP-Disjunction
\[(S \text{ wp } Q \lor R) = (S \text{ wp } Q) \lor (S \text{ wp } R)\]

- Does the disjunction rule hold?
  - \(\Leftarrow\) yes
  - \(\Rightarrow\) ???

- Might not hold if \(S\) non-deterministic
Skip and abort

⊢ skip \textit{wp} P = P

Dijkstra: "... cannot even do nothing in the sense of leaving things as they are;"
Skip and abort

⊢ skip \textbf{wp} \ P = \ P

⊢ abort \textbf{wp} \ P = \bot

- \textbf{abort} cannot even achieve the post condition \top
- Dijsktra: “… cannot even do nothing in the sense of leaving things as they are;”
\[ \vdash x \leftarrow e \uparrow \wp Q = [e/x]Q \]

- Same rule we had already, but this additionally specifies that the precondition obtained is the weakest
\[ \vdash \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end } \wp \ Q = \]
\[ (B \implies (S_1 \wp Q)) \land (\neg B \implies (S_2 \wp Q)) \]
Composition

⊢ \( S_1; S_2 \) wp \( Q = S_1 \) wp (\( S_2 \) wp \( Q \))
Composition

\[ \vdash S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q) \]

\[ \{P\} \ x \leftarrow x + 1; \ y \leftarrow x + y \ \{y > 17\} \], what is weakest \(P\)?
\[ \vdash S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q) \]

\( \{P\} \ x \leftarrow x + 1; \ y \leftarrow x + y \ \{y > 17\} \), what is weakest \( P \)?

\[ S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q) \]
Composition

\[ \vdash S_1; S_2 \text{ wp } Q = S_1 \text{ wp (} S_2 \text{ wp } Q \text{)} \]

\[ \{P\} \ x \leftarrow x + 1; \ y \leftarrow x + y \ \{y > 17\}, \text{ what is weakest } P? \]

\[ S_1; S_2 \text{ wp } Q = S_1 \text{ wp (} S_2 \text{ wp } Q \text{)} \]
\[ x \leftarrow x + 1; \ y \leftarrow x + y \ \text{ wp} \ \{y > 17\} \]
Composition

\[ \vdash S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q) \]

\{P\} x \leftarrow x + 1; y \leftarrow x + y \{y > 17\}, what is weakest \( P \)?

\[ S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q) \]
\[ x \leftarrow x + 1; y \leftarrow x + y \text{ wp } \{y > 17\} \]
\[ = x \leftarrow x + 1 \text{ wp } (y \leftarrow x + y \text{ wp } \{y > 17\}) \]
Composition

\[\vdash S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q)\]

\[
\{P\} \ x \leftarrow x + 1; \ y \leftarrow x + y \ \{y > 17\}, \text{ what is weakest } P? \]

\[
S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q) \]
\[
x \leftarrow x + 1; \ y \leftarrow x + y \ \text{ wp } \{y > 17\} \]
\[= \ x \leftarrow x + 1 \ \text{ wp } (y \leftarrow x + y \ \text{ wp } \{y > 17\})\]
\[= \ x \leftarrow x + 1 \ \text{ wp } \{x + y > 17\}\]
Composition

\[
\vdash S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q)
\]

\[
\{P\} \ x \leftarrow x + 1; \ y \leftarrow x + y \ \{y > 17\}, \text{ what is weakest } P?\]

\[
S_1; S_2 \text{ wp } Q = S_1 \text{ wp } (S_2 \text{ wp } Q)
\]

\[
x \leftarrow x + 1; \ y \leftarrow x + y \ \text{ wp } \{y > 17\}
= \ x \leftarrow x + 1 \ \text{ wp } (y \leftarrow x + y \ \text{ wp } \{y > 17\})
= \ x \leftarrow x + 1 \ \text{ wp } \{x + y > 17\}
= \ \{x + y + 1 > 17\}
Composition

⊢ \textit{S}_1; \textit{S}_2 \ wp \ Q = \textit{S}_1 \ wp (\textit{S}_2 \ wp \ Q)

\{P\} \ x \leftarrow x + 1; \ y \leftarrow x + y \ \{y > 17\}, \text{ what is weakest } P?

\textit{S}_1; \textit{S}_2 \ wp \ Q = \textit{S}_1 \ wp (\textit{S}_2 \ wp \ Q)
\ x \leftarrow x + 1; \ y \leftarrow x + y \ wp \ \{y > 17\}
= \ x \leftarrow x + 1 \ wp (y \leftarrow x + y \ wp \ \{y > 17\})
= \ x \leftarrow x + 1 \ wp \ \{x + y > 17\}
= \ \{x + y + 1 > 17\}
= \ \{x + y > 16\}
Consider the loop below and assume post-condition \( Q \)

\[
\text{while } B \text{ do } S \text{ end}
\]

If the loop is never entered, then the precondition must be

\[
P_0 \equiv \neg B \land Q
\]
Consider the loop below and assume post-condition $Q$

\[
\text{while } B \text{ do } S \text{ end}
\]

If the loop is never entered, then the precondition must be

\[
P_0 \equiv \neg B \land Q
\]

If executed exactly once the precondition must be

\[
P_1 \equiv B \land (S \ \text{wp} \ P_0)
\]
Loops

Consider the loop below and assume post-condition $Q$

```
while $B$ do $S$ end
```

- If the loop is never entered, then the precondition must be $P_0 \equiv \neg B \land Q$
- If executed exactly once the precondition must be $P_1 \equiv B \land (S \text{ wp } P_0)$
- If executed exactly twice the precondition must be $P_2 \equiv B \land (S \text{ wp } P_1)$
Inductively

\[ P_0 \equiv \neg B \land Q \]

\[ P_i \equiv B \land (S \ wp \ P_{i-1}) \]
Inductively

\[ P_0 \equiv \neg B \land Q \]

\[ P_i \equiv B \land (S \ wp \ P_{i-1}) \]

- What is the guarantee then that the loop terminates and that \( Q \) is true when it does?
Inductively

\[ P_0 \equiv \neg B \land Q \]

\[ P_i \equiv B \land (S \ wp\ P_{i-1}) \]

- What is the guarantee then that the loop terminates and that \( Q \) is true when it does?

- That there is some \( i \) s.t. \( P_i \) is true
while $B$ do $S$ end $\text{wp } Q \equiv \exists k. (k \geq 0 \land P_k)$

where

$$P_0 \equiv \neg B \land Q$$

$$P_i \equiv B \land (S \text{ wp } P_{i-1})$$
Loop’s weakest precondition

```
while B do S end wp Q ≡ ∃k.(k ≥ 0 ∧ P_k)
```

where

```
P_0 ≡ \neg B \land Q
```

```
P_i ≡ B \land (S \ wp \ P_{i-1})
```

- $P_k$ is the weakest precondition for the loop to terminate after exactly $k$ iterations in a state satisfying $Q$
- Loop’s weakest precondition is thus $P_0 \lor P_1 \lor P_2 \lor \ldots$
References

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